

Documentación de C#

Aprenda a escribir aplicaciones con el lenguaje de programación C# en la plataforma .NET.

Aprender a programar en C#

INTRODUCCIÓN

[Aprender C# | Tutoriales, cursos, vídeos y mucho más ↗](#)

VIDEO

[Serie de vídeos para principiantes de C# ↗](#)

TUTORIAL

[Tutoriales autoguiados](#)

[Tutorial en el explorador](#)

REFERENCIA

[C# en Q&A](#)

[Lenguajes en foros de la comunidad tecnológica de .NET ↗](#)

[C# en Stack Overflow ↗](#)

[C# en Discord ↗](#)

Aspectos básicos de C#

INFORMACIÓN GENERAL

[Un paseo por C#](#)

[Dentro de un programa de C#](#)

[Serie de vídeos de información destacada de C# ↗](#)

CONCEPTO

[Sistema de tipos](#)

[Programación orientada a objetos](#)

[Técnicas funcionales](#)

[Excepciones](#)

[Estilo de codificación](#)

TUTORIAL

[Mostrar línea de comandos](#)

[Introducción a las clases](#)

[C# orientado a objetos](#)

[Conversión de tipos](#)

[Detección de patrones](#)

[Uso de LINQ para consultar datos](#)

Novedades

NOVEDADES

[Novedades de C# 13](#)

[Novedades de C# 12](#)

[Novedades de C# 11](#)

[Novedades de C# 10](#)

TUTORIAL

[Exploración de los tipos de registros](#)

[Exploración de instrucciones de nivel superior](#)

[Exploración de nuevos patrones](#)

[Escritura de un controlador de interpolación de cadenas personalizado](#)

REFERENCIA

[Cambios importantes en el compilador de C#](#)

[Compatibilidad de versiones](#)

Conceptos clave

INFORMACIÓN GENERAL

[Estrategia del lenguaje C#](#)

[Conceptos de programación](#)

CONCEPTO

[Language-Integrated Query \(LINQ\)](#)

[Programación asíncrona](#)

Conceptos avanzados

REFERENCIA

[Reflexión y atributos](#)

[Árboles de expresión](#)

[Interoperabilidad nativa](#)

[Ingeniería de rendimiento](#)

[SDK de la plataforma del compilador de .NET](#)

Mantente en contacto

REFERENCIA

[Comunidad de desarrolladores de .NET ↗](#)

[YouTube ↗](#)

[Twitter ↗](#)

Paseo por el lenguaje C#

Artículo • 14/05/2024

El lenguaje C# es el lenguaje más popular para la [plataforma .NET](#), un entorno de desarrollo gratuito, multiplataforma y de código abierto. Los programas de C# se pueden ejecutar en muchos dispositivos diferentes, desde dispositivos de Internet de las cosas (IoT) a la nube y entre todas partes. Puede escribir aplicaciones para equipos y servidores móviles, de escritorio y portátiles.

C# es un lenguaje multiplataforma de uso general que hace que los desarrolladores sean productivos al escribir código de alto rendimiento. Con millones de desarrolladores, C# es el lenguaje .NET más popular. C# tiene una amplia compatibilidad con el ecosistema y todas las [cargas de trabajo](#) de .NET. Basado en principios orientados a objetos, incorpora muchas características de otros paradigmas, en particular de la programación funcional. Las características de bajo nivel admiten escenarios de alta eficiencia sin necesidad de escribir código no seguro. La mayor parte del entorno de ejecución y las bibliotecas de .NET se escriben en C#, y los avances en C# suelen beneficiar a todos los desarrolladores de .NET.

Hola a todos

El programa "Hola mundo" tradicionalmente se usa para presentar un lenguaje de programación. En este caso, se usa C#:

```
C#  
  
// This line prints "Hello, World"  
Console.WriteLine("Hello, World");
```

La línea a partir de `//` es un *comentario de una sola línea*. Los comentarios de una sola línea de C# comienzan por `//` y continúan hasta el final de la línea actual. C# también admite *comentarios de varias líneas*. Los comentarios de varias líneas comienzan con `/*` y terminan con `*/`. El método `WriteLine` de la clase `Console`, que se encuentra en el espacio de nombres `System`, genera la salida del programa. Esta clase la proporcionan las bibliotecas de clase estándar, a las que, de forma predeterminada, los programas de C# hacen referencia automáticamente.

En el ejemplo anterior se muestra una forma de un programa "Hello, World", mediante [instrucciones de nivel superior](#). Las versiones anteriores de C# requerían definir el punto de entrada del programa en un método. Este formato sigue siendo válido y lo verá en

muchos ejemplos de C# existentes. También debe estar familiarizado con este formulario, como se muestra en el ejemplo siguiente:

```
C#  
  
using System;  
  
class Hello  
{  
    static void Main()  
    {  
        // This line prints "Hello, World"  
        Console.WriteLine("Hello, World");  
    }  
}
```

Esta versión muestra los bloques de creación que usa en los programas. 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 varios 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`. En el ejemplo anterior, ese espacio de nombres se incluyó [implícitamente](#).

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. De manera predeterminada, cuando no hay instrucciones de nivel superior, un método estático denominado `Main` actúa como [punto de entrada](#) de un programa de C#.

Ambos formularios de punto de entrada generan código equivalente. Cuando se usan instrucciones de nivel superior, el compilador sintetiza la clase y el método contenedor para el punto de entrada del programa.

💡 Sugerencia

Los ejemplos de este artículo le ofrecen un primer vistazo al código de C#. Algunos ejemplos pueden mostrar elementos de C# con los que no está familiarizado. Cuando esté listo para aprender C#, comience con nuestros [tutoriales para principiantes](#) o profundice en los vínculos de cada sección. Si tiene experiencia en [Java](#), [JavaScript](#), [TypeScript](#) o [Python](#), lea nuestras sugerencias para ayudarle a encontrar la información que necesita para aprender rápidamente C#.

Características conocidas de C#

C# es accesible para principiantes, pero ofrece características avanzadas para desarrolladores experimentados que escriben aplicaciones especializadas. Puede ser productivo rápidamente. Puede obtener más información sobre técnicas especializadas a medida que las necesite para las aplicaciones.

Las aplicaciones de C# se benefician de la [administración automática de memoria](#) de .NET Runtime. Las aplicaciones de C# también usan las amplias [bibliotecas de runtime](#) proporcionadas por el SDK de .NET. Algunos componentes son independientes de la plataforma, como las bibliotecas del sistema de archivos, colecciones de datos y bibliotecas matemáticas. Otros son específicos de una sola carga de trabajo, como las bibliotecas web de ASP.NET Core o la biblioteca de interfaz de usuario de .NET MAUI. Un ecosistema de código abierto enriquecido en [NuGet](#) aumenta las bibliotecas que forman parte del entorno de ejecución. Estas bibliotecas proporcionan aún más componentes que puede usar.

C# es parte de la familia de lenguajes C. La [sintaxis de C#](#) le resultará conocida si ha usado C, C++, JavaScript o Java. Al igual que todos los lenguajes de la familia C, los punto y coma (;) definen el final de las instrucciones. Los identificadores de C# distinguen mayúsculas de minúsculas. C# tiene el mismo uso de llaves, { y }, instrucciones de control como `if`, `else` y `switch`, y construcciones de bucle como `for` y `while`. C# también tiene una instrucción `foreach` para cualquier tipo de colección.

C# es un lenguaje *fuertemente tipado*. Cada variable que declare tiene un tipo conocido en tiempo de compilación. El compilador o las herramientas de edición indican si usa ese tipo incorrectamente. Puede corregir esos errores antes de ejecutar el programa. Los [tipos de datos fundamentales](#) están integrados en el lenguaje y el entorno de ejecución: tipos de valor como `int`, `double`, `char`, tipos de referencia como `string`, matrices y otras colecciones. A medida que escribe sus programas, crea sus propios tipos. Esos tipos pueden ser tipos `struct` para valores o tipos `class` que definen el comportamiento orientado a objetos. Puede agregar el modificador `record` a los tipos `struct` o `class` para que el compilador sintetice el código para las comparaciones de

igualdad. También puede crear definiciones `interface`, que definen un contrato o un conjunto de miembros, que debe proporcionar un tipo que implemente esa interfaz. También puede definir tipos y métodos genéricos. Los `genéricos` usan *parámetros de tipo* para proporcionar un marcador de posición para un tipo real cuando se usa.

A medida que escribe código, se definen funciones, también denominadas `métodos`, como miembros de tipos `struct` y `class`. Estos métodos definen el comportamiento de los tipos. Los métodos se pueden sobrecargar, con un número o tipos diferentes de parámetros. Opcionalmente, los métodos pueden devolver un valor. Además de los métodos, los tipos de C# pueden tener `propiedades`, que son elementos de datos respaldados por funciones denominadas *descriptores de acceso*. Los tipos de C# pueden definir `eventos`, que permiten que un tipo notifique a los suscriptores de acciones importantes. C# admite técnicas orientadas a objetos como herencia y polimorfismo para tipos `class`.

C# usa `excepciones` para notificar y controlar errores. Estará familiarizado con esta práctica si ha usado C++ o Java. El código produce una excepción cuando no puede hacer lo previsto. Otro código, independientemente del número de niveles de la pila de llamadas, puede recuperarse opcionalmente mediante un bloque `try - catch`.

Características distintivas de C#

Es posible que algunos elementos de C# le resulten menos familiares. `Language Integrated Query (LINQ)` proporciona una sintaxis común basada en patrones para consultar o transformar cualquier colección de datos. LINQ unifica la sintaxis para consultar colecciones en memoria, datos estructurados como XML o JSON, almacenamiento de base de datos e incluso API de datos basadas en la nube. Aprenderá un conjunto de sintaxis y podrá buscar y manipular datos independientemente de su almacenamiento. La consulta siguiente busca todos los alumnos cuyo promedio de calificaciones es mayor que 3,5:

```
C#  
  
var honorRoll = from student in Students  
                 where student.GPA > 3.5  
                 select student;
```

La consulta anterior funciona para muchos tipos de almacenamiento representados por `Students`. Podría ser una colección de objetos, una tabla de base de datos, un blob de almacenamiento en la nube o una estructura XML. La misma sintaxis de consulta funciona para todos los tipos de almacenamiento.

El [modelo de programación asincrónica basado en tareas](#) permite escribir código que se lee como si se ejecutara sincrónicamente, aunque se ejecuta de forma asincrónica. Utiliza las palabras clave `async` y `await` para describir métodos asincrónicos y cuando una expresión se evalúa de forma asincrónica. En el ejemplo siguiente, se espera una solicitud web asincrónica. Cuando se completa la operación asincrónica, el método devuelve la longitud de la respuesta:

C#

```
public static async Task<int> GetPageLengthAsync(string endpoint)
{
    var client = new HttpClient();
    var uri = new Uri(endpoint);
    byte[] content = await client.GetByteArrayAsync(uri);
    return content.Length;
}
```

C# también admite una instrucción `await foreach` para iterar una colección respaldada por una operación asincrónica, como una API de paginación de GraphQL. En el ejemplo siguiente, se leen datos en fragmentos y se devuelve un iterador que proporciona acceso a cada elemento cuando está disponible:

C#

```
public static async IAsyncEnumerable<int> ReadSequence()
{
    int index = 0;
    while (index < 100)
    {
        int[] nextChunk = await GetNextChunk(index);
        if (nextChunk.Length == 0)
        {
            yield break;
        }
        foreach (var item in nextChunk)
        {
            yield return item;
        }
        index++;
    }
}
```

Los autores de llamadas pueden iterar la colección mediante una instrucción `await foreach`:

C#

```
await foreach (var number in ReadSequence())
{
    Console.WriteLine(number);
}
```

C# proporciona [coincidencia de patrones](#). Esas expresiones permiten inspeccionar los datos y tomar decisiones en función de sus características. La coincidencia de patrones proporciona una excelente sintaxis para el flujo de control en función de los datos. En el código siguiente se muestra cómo se pueden expresar los métodos para las operaciones booleanas *and*, *or* y *xor* mediante la sintaxis de coincidencia de patrones:

C#

```
public static bool Or(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
    };

public static bool And(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => false,
        (false, true) => false,
        (false, false) => false,
    };
public static bool Xor(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => false,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
    };
}
```

Las expresiones de coincidencia de patrones se pueden simplificar mediante `_` como captura para cualquier valor. En el ejemplo siguiente, se muestra cómo se puede simplificar el método *and*:

C#

```
public static bool ReducedAnd(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
```

```
    (_, _) => false,  
};
```

Por último, como parte del ecosistema de .NET, puede usar [Visual Studio](#) o [Visual Studio Code](#) con [C# DevKit](#). Estas herramientas proporcionan una comprensión integral de C#, incluido el código que se debe escribir. También proporcionan funcionalidades de depuración.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Plan de desarrollo para desarrolladores de Java que aprenden C#

Artículo • 11/04/2024

C# y Java tienen muchas similitudes. A medida que aprende C#, puede aplicar gran parte del conocimiento que ya tiene desde la programación en Java:

1. **Sintaxis similar:** Java y C# están en la familia de lenguajes C. Esta similitud significa que ya puede leer y comprender C#. Hay algunas diferencias, pero la mayoría de la sintaxis es la misma que Java y C. Las llaves y puntos y comas son familiares. Las instrucciones de control como `if`, `else`, `switch` son las mismas. Las instrucciones de bucle de `for`, `while` y `do...while` son las mismas. Las mismas palabras clave para `class` y `interface` están en ambos idiomas. Los modificadores de acceso de `public` a `private` son los mismos. Incluso muchos de los tipos integrados usan las mismas palabras clave: `int`, `string` y `double`.
2. **Paradigma orientado a objetos:** Java y C# son lenguajes orientados a objetos. Los conceptos de polimorfismo, abstracción y encapsulación se aplican en ambos lenguajes. Ambos agregaron nuevas construcciones, pero las características principales siguen siendo relevantes.
3. **Fuertemente tipado:** Java y C# son lenguajes fuertemente tipados. El tipo de datos de las variables se declara explícita o implícitamente. El compilador aplica la seguridad de tipos. El compilador detecta errores relacionados con tipos en el código, antes de ejecutar el código.
4. **Multiplataforma:** tanto Java como C# son multiplataforma. Puede ejecutar las herramientas de desarrollo en su plataforma favorita. La aplicación se puede ejecutar en varias plataformas. La plataforma de desarrollo no es necesaria para que coincida con la plataforma de destino.
5. **Control de excepciones:** Java y C# inicián excepciones para indicar errores. Ambos usan bloques `try` - `catch` - `finally` para controlar las excepciones. Las clases Excepción tienen nombres y jerarquías de herencia similares. Una diferencia es que C# no tiene el concepto de *excepciones comprobadas*. Cualquier método podría (en teoría) producir cualquier excepción.
6. **Bibliotecas estándar:** el entorno de ejecución de .NET y la biblioteca estándar de Java (JSL) admiten tareas comunes. Ambos tienen ecosistemas extensos para otros paquetes de código abierto. En C#, el administrador de paquetes es [NuGet](#). Es análogo a Maven.
7. **Recolección de elementos no utilizados:** ambos lenguajes emplean la administración automática de memoria a través de la recolección de elementos no utilizados. El tiempo de ejecución reclama la memoria de los objetos a los que no

se hace referencia. Una diferencia es que C# permite crear tipos de valor, como tipos `struct`.

Será productivo en C# casi inmediatamente debido a las similitudes. A medida que avance, debe aprender características y expresiones en C# que no están disponibles en Java:

1. **Coincidencia de patrones**: la coincidencia de patrones permite instrucciones condicionales y expresiones concisas basadas en la forma de estructuras de datos complejas. La instrucción `is` comprueba si una variable "es" algún patrón. La expresión `switch basada en patrones` proporciona una rica sintaxis para inspeccionar una variable y tomar decisiones basadas en sus características.
2. **Interpolación de cadenas y literales de cadena**: la interpolación de cadenas le permite insertar expresiones evaluadas en una cadena, en lugar de utilizar identificadores posicionales. Los literales de cadena sin formato proporcionan una manera de minimizar las secuencias de escape en el texto.
3. **Tipos anulables y no anulables**: C# admite tipos de valor *anulables*, y tipos de referencia *anulables* añadiendo el sufijo `?` a un tipo. Para los tipos anulables, el compilador le avisa si no comprueba `null` antes de dereferenciar la expresión. Para los tipos no anulables, el compilador le avisa si puede estar asignando un valor `null` a esa variable. Los tipos de referencia no anulables minimizan los errores de programación que lanzan un `System.NullReferenceException`.
4. **Métodos de extensión**: en C#, puede crear métodos que *amplíen* una clase o interfaz. Los métodos de extensión amplían el comportamiento de un tipo desde una biblioteca o todos los tipos que implementan una interfaz determinada.
5. **LINQ**: Language Integrated Query (LINQ) proporciona una sintaxis común para consultar y transformar datos, independientemente de su almacenamiento.
6. **Funciones locales**: en C#, puede anidar funciones dentro de métodos u otras funciones locales. Las funciones locales proporcionan otra capa de encapsulación.

Hay otras características de C# que no están en Java. Verá características como las instrucciones `async` y `await` y `using` para liberar automáticamente recursos que no son de memoria.

También hay algunas características similares entre C# y Java que tienen diferencias sutiles pero importantes:

1. **Propiedades e Indexadores**: tanto las propiedades como los indexadores (tratando una clase como una matriz o diccionario) tienen compatibilidad con lenguajes. En Java, son convenciones de nomenclatura para los métodos a partir de `get` y `set`.
2. **Registros**: en C#, los registros pueden ser `class` (referencia) o `struct` (valor). Los registros de C# pueden ser inmutables, pero no son necesarios para ser

inmutables.

3. **Tuplas** tienen una sintaxis diferente en C# y Java.
4. **Atributos** son similares a las anotaciones de Java.

Por último, hay características del lenguaje Java que no están disponibles en C#:

1. **Excepciones comprobadas**: en C#, cualquier método podría producir teóricamente cualquier excepción.
2. **Covarianza de matriz comprobada**: en C#, las matrices no son covariantes de forma segura. Debe usar las clases e interfaces de colección genéricas si necesita estructuras covariantes.

En general, el aprendizaje de C# para un desarrollador experimentado en Java debe ser suave. Encontrará suficientes expresiones conocidas para ser productivos rápidamente y aprenderá las nuevas expresiones rápidamente.

Plan de desarrollo para desarrolladores de JavaScript y TypeScript que aprenden C#

Artículo • 16/04/2024

C#, TypeScript y JavaScript son miembros de la familia de lenguajes de C. Las similitudes entre los lenguajes le ayudan a ser productivos rápidamente en C#.

1. **Sintaxis similar:** JavaScript, TypeScript y C# están en la familia de lenguajes C. Esta similitud significa que ya puede leer y comprender C#. Hay algunas diferencias, pero la mayoría de la sintaxis es la misma que JavaScript y C. Las llaves y puntos y comas son familiares. Las instrucciones de control como `if`, `else` y `switch` son las mismas. Las instrucciones de bucle de `for`, `while` y `do...while` son las mismas. Las mismas palabras clave para `class` y `interface` están en C# y TypeScript. Los modificadores de acceso de TypeScript y C#, de `public` a `private`, son los mismos.
2. **El token `=>`:** todos los lenguajes admiten definiciones de función ligeras. En C# se denominan *expresiones lambda*, mientras que en JavaScript normalmente se denominan *funciones de flecha*.
3. **Jerarquías de funciones:** los tres lenguajes admiten *funciones locales*, que son funciones definidas en otras funciones.
4. **Async y Await:** los tres lenguajes comparten las mismas palabras clave `async` y `await` para la programación asíncrona.
5. **Recolección de elementos no utilizados:** los tres lenguajes dependen de un recolector de elementos no utilizados para la administración automática de memoria.
6. **Modelo de eventos:** la sintaxis `event` de C# es similar al modelo de JavaScript para eventos del modelo de objetos de documento (DOM).
7. **Administrador de paquetes:** [NuGet](#) es el administrador de paquetes más común para C# y .NET, similar a npm para las aplicaciones de JavaScript. Las bibliotecas de C# se entregan en [ensamblados](#).

A medida que siga aprendiendo C#, aprenderá conceptos que no forman parte de JavaScript. Algunos de estos conceptos pueden resultarle familiares si usa TypeScript:

1. **Sistema de tipos de C#:** C# es un lenguaje fuertemente tipado. Cada variable tiene un tipo y ese tipo no puede cambiar. Debe definir tipos `class` o `struct`. Puede definir definiciones `interface` que definen el comportamiento implementado por otros tipos. TypeScript incluye muchos de estos conceptos, pero como TypeScript se basa en JavaScript, el sistema de tipos no es tan estricto.

2. **Coincidencia de patrones:** la coincidencia de patrones permite instrucciones condicionales y expresiones concisas basadas en la forma de estructuras de datos complejas. La expresión `is` comprueba si una variable "es" algún patrón. La expresión `switch basada en patrones` proporciona una rica sintaxis para inspeccionar una variable y tomar decisiones basadas en sus características.
3. **Interpolación de cadenas y literales de cadena:** la interpolación de cadenas le permite insertar expresiones evaluadas en una cadena, en lugar de utilizar identificadores posicionales. Los literales de cadena sin formato proporcionan una manera de minimizar las secuencias de escape en el texto.
4. **Tipos anulables y no anulables:** C# admite tipos de valor *anulables*, y tipos de referencia *anulables* añadiendo el sufijo `?` a un tipo. Para los tipos anulables, el compilador le avisa si no comprueba `null` antes de dereferenciar la expresión. Para los tipos no anulables, el compilador le avisa si puede estar asignando un valor `null` a esa variable. Estas características pueden minimizar la aplicación iniciando una `System.NullReferenceException`. La sintaxis puede resultar familiar por el uso que hace TypeScript de `?` para las propiedades opcionales.
5. **LINQ:** Language Integrated Query (LINQ) proporciona una sintaxis común para consultar y transformar datos, independientemente de su almacenamiento.

A medida que se aprende más se aprecian otras diferencias, pero muchas de ellas son de menor alcance.

Algunas características y expresiones conocidas de JavaScript y TypeScript no están disponibles en C#:

1. **tipos dinámicos:** C# usa la escritura estática. Una declaración de variable incluye el tipo y ese tipo no puede cambiar. Hay un tipo `dynamic` en C# que proporciona un enlace en tiempo de ejecución.
2. **Herencia prototípica:** la herencia de C# forma parte de la declaración de tipos. Una declaración `class` de C# indica cualquier clase base. En JavaScript, puede establecer la propiedad `__proto__` para establecer el tipo base en cualquier instancia.
3. **Lenguaje interpretado:** el código de C# debe compilarse antes de ejecutarlo. El código JavaScript se puede ejecutar directamente en el explorador.

Además, algunas características más de TypeScript no están disponibles en C#:

1. **Tipos de unión:** C# no admite tipos de unión. Sin embargo, las propuestas de diseño están en curso.
2. **Decoradores:** C# no tiene decoradores. Algunos decoradores comunes, como `@sealed`, son palabras clave reservadas en C#. Otros decoradores comunes pueden

tener [atributos](#) correspondientes. Para otros decoradores, puede crear sus propios atributos.

3. *Sintaxis más indulgente*: el compilador de C# analiza el código de forma más estricta que JavaScript.

Si va a compilar una aplicación web, considere la posibilidad de usar [Blazor](#) para compilar la aplicación. Blazor es un marco web de pila completa creado para .NET y C#. Los componentes de Blazor se pueden ejecutar en el servidor, como ensamblados de .NET o en el cliente mediante WebAssembly. Blazor admite la interoperabilidad con sus bibliotecas favoritas de JavaScript o TypeScript.

Plan de desarrollo para desarrolladores de Python que aprenden C#

Artículo • 16/04/2024

C# y Python comparten conceptos similares. Estas construcciones conocidas le ayudan a aprender C# cuando ya conoce Python.

1. **Orientados a objetos:** tanto Python como C# son lenguajes orientados a objetos. Todos los conceptos relacionados con las clases de Python se aplican en C#, incluso si la sintaxis es diferente.
2. **Multiplataforma:** tanto Python como C# son lenguajes multiplataforma. Las aplicaciones escritas en ambos lenguajes se pueden ejecutar en muchas plataformas.
3. **Recolección de elementos no utilizados:** ambos lenguajes emplean la administración automática de memoria a través de la recolección de elementos no utilizados. El runtime reclama la memoria de los objetos a los que no se hace referencia.
4. **Fuertemente tipado:** Python y C# son lenguajes fuertemente tipados. La coerción de tipos no se produce implícitamente. Hay diferencias que se describen más adelante, ya que C# se escribe estáticamente, mientras que Python se escribe dinámicamente.
5. **Async y Await:** la característica `async` y `await` de Python se inspiró directamente en la compatibilidad con `async` y `await` de C#.
6. **Coincidencia de patrones:** la expresión `match` de Python y la coincidencia de patrones es similar a la expresión `switch` de [coincidencia de patrones](#) de C#. Los usa para inspeccionar una expresión de datos compleja para determinar si coincide con un patrón.
7. **Palabras clave de la instrucción:** Python y C# comparten muchas palabras clave, como `if`, `else`, `while`, `for` y muchas otras. Aunque no toda la sintaxis es la misma, hay suficiente similitud como para que pueda leer C# si sabe Python.

A medida que empiece a aprender C#, aprenderá estos conceptos importantes en los que C# se diferencia de Python:

1. **Sangría frente a tokens:** en Python, las líneas nuevas y la sangría son elementos sintácticos de primera clase. En C#, el espacio en blanco no es significativo. Los tokens como `;` separan instrucciones, y los tokens `{` y `}` controlan el ámbito de bloque de `if` y otras instrucciones de bloque. Sin embargo, para mejorar la legibilidad, la mayoría de los estilos de codificación (incluido el estilo usado en

estos documentos) usan sangría para reforzar los ámbitos de bloque declarados por `{` y `}`.

2. **Escritura estática**: en C#, una declaración de variable incluye su tipo. La reasignación de una variable a un objeto de un tipo diferente genera un error del compilador. En Python, el tipo puede cambiar cuando se reasigna.
3. **Tipos que aceptan valores NULL**: las variables de C# pueden *admitir un valor NULL o no aceptar valores NULL*. Un tipo que no acepta valores NULL es uno que no puede ser NULL (o nada). Siempre hace referencia a un objeto válido. Por el contrario, un tipo que acepta valores NULL puede hacer referencia a un objeto válido o null.
4. **LINQ**: las palabras clave de expresión de consulta que componen Language Integrated Query (LINQ) no son palabras clave en Python. Sin embargo, las bibliotecas de Python como `itertools`, `more-itertools` y `py-linq` proporcionan una funcionalidad similar.
5. **Genéricos**: los genéricos de C# usan la escritura estática de C# para realizar aserciones sobre los argumentos proporcionados para los parámetros de tipo. Es posible que un algoritmo genérico necesite especificar restricciones que un tipo de argumento debe satisfacer.

Por último, hay algunas características de Python que no están disponibles en C#:

1. **(Duck) typing estructural**: en C#, los tipos tienen nombres y declaraciones. A excepción de **tuplas**, los tipos con la misma estructura no son intercambiables.
2. **REPL**: C# no tiene un read–eval–print loop (REPL) para crear rápidamente prototipos de soluciones.
3. **Espacio en blanco significativo**: debe usar correctamente llaves `{` y `}` para anotar el ámbito del bloque.

Aprender C# si sabe Python es un recorrido sencillo. Ambos lenguajes tienen conceptos y expresiones similares.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

nuestra guía para
colaboradores.

Anotaciones sobre la estrategia de C#

Artículo • 09/05/2023

Seguiremos desarrollando C# para satisfacer las necesidades cambiantes de los desarrolladores y para que continúe siendo un lenguaje de programación de vanguardia. Innovaremos con entusiasmo y ampliamente en colaboración con los equipos responsables de las bibliotecas de .NET, las herramientas de desarrollo y la compatibilidad con cargas de trabajo, al tiempo que tendremos cuidado de mantener el espíritu del lenguaje. Teniendo en cuenta la diversidad de dominios en los que se usa C#, daremos preferencia a las mejoras del lenguaje y el rendimiento que beneficien a todos o a la mayoría de los desarrolladores y que mantengan un alto compromiso con la compatibilidad con versiones anteriores. Seguiremos aportando más capacidad al amplio ecosistema de .NET y ampliando su rol en el futuro de C#, al tiempo que mantendremos el control sobre las decisiones de diseño.

Estrategia que guía C#

La estrategia de C# guía nuestras decisiones sobre la evolución del lenguaje, y estas anotaciones proporcionan información sobre nuestra opinión respecto a algunas afirmaciones importantes.

"innovaremos con entusiasmo y ampliamente"

La comunidad de C# sigue creciendo y el lenguaje continúa evolucionando para satisfacer las necesidades y expectativas de la comunidad. Nos inspiramos en una gran variedad de orígenes para seleccionar características que benefician a un amplio segmento de desarrolladores de C# y que proporcionan mejoras coherentes en cuanto a productividad, legibilidad y rendimiento.

"tendremos cuidado de mantener el espíritu del lenguaje"

Evaluamos las nuevas ideas teniendo en cuenta el espíritu y la historia del lenguaje C#. Damos prioridad a las innovaciones que sean útiles para la mayoría de los desarrolladores de C# actuales.

"mejoras que beneficien a todos o a la mayoría de los desarrolladores"

Los desarrolladores usan C# en todas las cargas de trabajo de .NET, como front-ends y back-ends web, desarrollo de aplicaciones nativas de nube, desarrollo de aplicaciones

de escritorio y creación de aplicaciones multiplataforma. Nos centramos en las nuevas características que tienen el mayor impacto, ya sea directamente o potenciando mejoras en las bibliotecas comunes. El desarrollo de características del lenguaje incluye la integración en nuestras herramientas de desarrollo y recursos de aprendizaje.

"alto compromiso con la compatibilidad con versiones anteriores"

Respetamos el hecho de que actualmente hay una gran cantidad de código de C# en uso. Cualquier posible cambio importante se analiza detenidamente en cuanto a la escala y el impacto de la interrupción en la comunidad de C#.

"mantendremos el control"

El [diseño del lenguaje C#](#) es abierto, con la participación de la comunidad. Cualquier persona puede proponer nuevas características de C# en nuestros [repositorios de GitHub](#). El [equipo de diseño del lenguaje](#) toma las decisiones finales después de ponderar la entrada de la comunidad.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Introducción a C#

Artículo • 11/04/2024

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.

<https://docs.microsoft.com/shows/CSharp-101/What-is-C/player>

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 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.

101 ejemplos de LINQ ↗

Este ejemplo requiere la herramienta global [dotnet-try](#) ↗. Una vez que instale la herramienta y clone el repositorio [try-samples](#) ↗, puede aprender Language Integrated Query (LINQ) mediante un conjunto de 101 ejemplos que puede ejecutar de forma interactiva. Puede descubrir diferentes maneras de consultar, explorar y transformar secuencias de datos.

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

Proporcionar comentarios sobre el producto 

Configuración de un entorno local

Artículo • 10/05/2023

El primer paso en la ejecución de un tutorial en el equipo es configurar un entorno de desarrollo.

- Se recomienda tener [Visual Studio](#) para Windows o Mac. Puede descargar una versión gratuita desde la [página de descargas de Visual Studio](#). Visual Studio incluye el SDK de .NET.
- También se puede usar el editor de [Visual Studio Code](#). Deberá instalar el [SDK de .NET](#) más reciente por separado.
- Si prefiere usar otro editor, deberá instalar el [SDK de .NET](#) más reciente.

Flujo de desarrollo de aplicaciones básico

Las instrucciones de estos tutoriales asumen que está usando la CLI de .NET para crear, compilar y ejecutar aplicaciones. Usará los comandos siguientes:

- [dotnet new](#) crea una aplicación. 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.
- [dotnet build](#) compila el archivo ejecutable.
- [dotnet run](#) ejecuta el archivo ejecutable.

Si usa Visual Studio 2019 para estos tutoriales, elegirá una selección de menú de Visual Studio cuando un tutorial le indique que ejecute uno de estos comandos de la CLI:

- **Archivo>Nuevo>Proyecto** crea una aplicación.
 - Se recomienda la plantilla de proyecto `Console Application`.
 - Se le ofrecerá la posibilidad de especificar una plataforma de destino. Los tutoriales siguientes funcionan mejor cuando el destino es .NET 5 o versiones posteriores.
- **Compilar>Compilar solución** crea el archivo ejecutable.
- **Depurar>Iniciar sin depurar** ejecuta 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.

Uso de números enteros y de punto flotante en C#

Artículo • 10/05/2023

En este tutorial se explican los tipos numéricos en C#. 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#.

💡 Sugerencia

Para pegar un fragmento de código dentro del **modo de enfoque** debe usar el método abreviado de teclado (`ctrl + v` o `cmd + v`).

Requisitos previos

En el tutorial se espera que tenga una máquina configurada para el desarrollo local. Consulte [Configuración del entorno local](#) para obtener instrucciones de instalación e información general sobre el desarrollo de aplicaciones en .NET.

Si no quiere configurar un entorno local, consulte la [versión interactiva en el explorador de este tutorial](#).

Análisis de las operaciones matemáticas con enteros

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

CLI de .NET

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

ⓘ Importante

Las plantillas de C# para .NET 6 usan *instrucciones de nivel superior*. Es posible que la aplicación no coincida con el código de este artículo si ya ha actualizado a

.NET 6. Para obtener más información, consulte el artículo [Las nuevas plantillas de C# generan instrucciones de nivel superior](#).

El SDK de .NET 6 también agrega un conjunto de directivas *implícitas* global using para proyectos que usan los SDK siguientes:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Estas directivas de global using implícitas incluyen los espacios de nombres más comunes para el tipo de proyecto.

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

```
C#  
  
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`:

```
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.

💡 Sugerencia

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. Un método es una serie de instrucciones agrupadas a la que se ha puesto un nombre. Llame a un método escribiendo el nombre del método seguido de `()`. La organización del código en métodos facilita empezar a trabajar con un ejemplo nuevo. Cuando termine, el código debe tener un aspecto similar al siguiente:

```
C#  
  
WorkWithIntegers();  
  
void WorkWithIntegers()  
{  
    int a = 18;  
    int b = 6;  
    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);  
}
```

La línea `WorkWithIntegers();` invoca el método. El código siguiente declara el método y lo define.

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:

C#

```
//WorkWithIntegers();
```

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. Dado que `WorkWithIntegers()` es un método, solo tiene que comentar una línea.

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 tras la llamada a `dotnet run` y la ejecución de `WorkWithIntegers()`:

C#

```
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:

C#

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

Combine muchas operaciones distintas para indagar más. Agregue algo similar a las líneas siguientes. Pruebe `dotnet run` de nuevo.

C#

```
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 este código:

C#

```
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`. El código debería tener este aspecto:

C#

```
// WorkWithIntegers();  
OrderPrecedence();  
  
void WorkWithIntegers()  
{  
    int a = 18;  
    int b = 6;  
    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);
}

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);
}

```

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 tras la llamada de método a `OrderPrecedence()`:

C#

```

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 para ver esos límites:

C#

```
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 para ver un ejemplo:

C#

```
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. Antes de comenzar la siguiente sección, mueva 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:

C#

```
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:

C#

```
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:

C#

```
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:

C#

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

Sabe que la repetición de 0.3 un número finito de veces 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:

C#

```
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:

C#

```
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`.

ⓘ Nota

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 círculo 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](#)

Instrucciones y bucles de C# `if`: tutorial de lógica condicional

Artículo • 10/05/2023

En este tutorial se enseña a escribir código de C# 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#.

💡 Sugerencia

Para pegar un fragmento de código dentro del **modo de enfoque** debe usar el método abreviado de teclado (`ctrl + v` o `cmd + v`).

Requisitos previos

En el tutorial se espera que tenga una máquina configurada para el desarrollo local. Consulte [Configuración del entorno local](#) para obtener instrucciones de instalación e información general sobre el desarrollo de aplicaciones en .NET.

Si prefiere ejecutar el código sin tener que configurar un entorno local, consulte la [versión interactiva en el explorador de este tutorial](#).

Toma de decisiones con la instrucción `if`.

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

CLI de .NET

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

ⓘ Importante

Las plantillas de C# para .NET 6 usan *instrucciones de nivel superior*. Es posible que la aplicación no coincida con el código de este artículo si ya ha actualizado a

.NET 6. Para obtener más información, consulte el artículo [Las nuevas plantillas de C# generan instrucciones de nivel superior](#).

El SDK de .NET 6 también agrega un conjunto de directivas *implícitas* global using para proyectos que usan los SDK siguientes:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Estas directivas de global using implícitas incluyen los espacios de nombres más comunes para el tipo de proyecto.

Este comando crea una nueva aplicación de consola de .NET en el directorio actual. Abra *Program.cs* en su editor favorito y reemplace el contenido por el código siguiente:

C#

```
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 "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:

C#

```
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.

💡 Sugerencia

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 una rama `else`. Agregue las dos últimas líneas del código siguiente (ya debe tener los cuatro primeros):

C#

```
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.

ⓘ Importante

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:

C#

```
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");
}
```

💡 Sugerencia

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 después del que ha escrito hasta ahora:

C#

```
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:

C#

```
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. Coloque el código existente en un método denominado `ExploreIf()`. Llámelo desde la parte superior del programa. Cuando termine, el código debe tener un aspecto similar al siguiente:

C#

```
ExploreIf();

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");
}
}
```

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

C#

```
//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. Agregue este código después de la llamada a `ExploreIf`:

C#

```
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`. Comprueba repetidamente la condición y ejecuta esas instrucciones hasta que la condición es 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`.

ⓘ Importante

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:

C#

```
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:

C#

```
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:

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

Otro bucle puede generar las columnas:

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

Puede anidar un bucle dentro de otro para formar pares:

```
C#
```

```
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. Cuando haya terminado, coloque el código de esta sección en un método denominado `ExploreLoops()`.

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 de selección](#)
- [Instrucciones de iteración](#)

Aprenda a administrar colecciones de datos mediante List<T> en C#

Artículo • 10/05/2023

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

Requisitos previos

En el tutorial se espera que tenga una máquina configurada para el desarrollo local. Consulte [Configuración del entorno local](#) para obtener instrucciones de instalación e información general sobre el desarrollo de aplicaciones en .NET.

Si prefiere ejecutar el código sin tener que configurar un entorno local, consulte la [versión interactiva en el explorador de este tutorial](#).

Un ejemplo de lista básica

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

Importante

Las plantillas de C# para .NET 6 usan *instrucciones de nivel superior*. Es posible que la aplicación no coincida con el código de este artículo si ya ha actualizado a .NET 6. Para obtener más información, consulte el artículo [Las nuevas plantillas de C# generan instrucciones de nivel superior](#).

El SDK de .NET 6 también agrega un conjunto de directivas *implícitas* `global using` para proyectos que usan los SDK siguientes:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Estas directivas de `global using` implícitas incluyen los espacios de nombres más comunes para el tipo de proyecto.

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

C#

```
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 MAYÚSCULAS. 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 al final del programa:

C#

```
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:

```
C#
```

```
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 programa:

```
C#
```

```
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 a la parte inferior del programa:

```
C#
```

```
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 a la parte inferior del programa:

```
C#  
  
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. Coloque todo el código que ha escrito en un nuevo método denominado `WorkWithStrings()`. Llame a ese método en la parte superior del programa. Cuando termine, el código debe tener un aspecto similar al siguiente:

```
C#  
  
WorkWithStrings();  
  
void WorkWithStrings()  
{  
    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 al programa después de llamar a `WorkWithStrings()`:

C#

```
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:

```
C#
```

```
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.

💡 Sugerencia

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

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 seguir estos tutoriales [adicionales](#) en su propio entorno de desarrollo.

Puede obtener más información sobre cómo trabajar con el tipo `List` en el artículo de los aspectos básicos de .NET que trata sobre las [colecciones](#). Ahí también podrá conocer muchos otros tipos de colecciones.

Estructura general de un programa de C#

Artículo • 07/04/2023

Los programas de C# constan de uno o más archivos. Cada archivo contiene cero o más espacios de nombres. Un espacio de nombres contiene tipos como clases, estructuras, interfaces, enumeraciones y delegados, u otros espacios de nombres. El siguiente ejemplo es el esqueleto de un programa de C# que contiene todos estos elementos.

```
C#  
  
// A skeleton of a C# program  
using System;  
  
// Your program starts here:  
Console.WriteLine("Hello world!");  
  
namespace YourNamespace  
{  
    class YourClass  
    {  
    }  
  
    struct YourStruct  
    {  
    }  
  
    interface IYourInterface  
    {  
    }  
  
    delegate int YourDelegate();  
  
    enum YourEnum  
    {  
    }  
  
    namespace YourNestedNamespace  
    {  
        struct YourStruct  
        {  
        }  
    }  
}
```

En el ejemplo anterior se usan *instrucciones de nivel superior* para el punto de entrada del programa. Esta característica se agregó en C# 9. Antes de C# 9, el punto de entrada

era un método estático denominado `Main`, como se muestra en el ejemplo siguiente:

C#

```
// 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 Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}
```

Secciones relacionadas

Obtenga información sobre estos elementos del programa en la sección de [tipos](#) de la guía de aspectos básicos:

- [Clases](#)
- [Structs](#)

- [Espacios de nombres](#)
- [Interfaces](#)
- [Enumeraciones](#)
- [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#.

Main() y argumentos de línea de comandos

Artículo • 26/06/2024

El método `Main` es el punto de entrada de una aplicación de C#. 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 tenga un método `Main`, deberá compilar el programa con la opción del compilador `StartupObject` para especificar qué método `Main` quiere utilizar como punto de entrada. Para obtener más información, consulte [StartupObject \(opciones del compilador de C#\)](#).

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

También puede usar instrucciones de nivel superior en un archivo como punto de entrada para la aplicación. Al igual que el método `Main`, las instrucciones de nivel superior también pueden [devolver valores](#) y acceder a los [argumentos de la línea de comandos](#). Para más información, consulte [Instrucciones de nivel superior](#).

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("The following arguments are passed:");

// Display the command line arguments using the args variable.
foreach (var arg in args)
{
    builder.AppendLine($"Argument={arg}");
}

Console.WriteLine(builder.ToString());

// Return a success code.
return 0;
```

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` debe declararse dentro de una clase o estructura. La `class` envolvente puede ser `static`.
- `Main` debe ser `static`.
- `Main` puede tener cualquier **modificador de acceso** (excepto `file`).
- `Main` puede tener un tipo de valor devuelto `void`, `int`, `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`. Esto excluye específicamente 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 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()`.

En la lista siguiente, se muestran las declaraciones `Main` más comunes:

C#

```
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
static async Task Main() { }
static async Task<int> Main() { }
static async Task Main(string[] args) { }
static async Task<int> Main(string[] args) { }
```

Los ejemplos anteriores no especifican un modificador de acceso, por lo que son implícitamente `private` de forma predeterminada. Esto es típico, pero es posible especificar cualquier modificador de acceso explícito.

💡 Sugerencia

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 asíncronas en `Main`.

Valores devueltos `Main()`

Puede devolver un objeto `int` desde el método `Main` si lo define de una de las siguientes maneras:

 Expandir tabla

declaración <code>Main</code>	Código del método <code>Main</code>
<code>static int Main()</code>	No se usa <code>args</code> ni <code>await</code>
<code>static int Main(string[] args)</code>	Se usa <code>args</code> , no se usa <code>await</code>
<code>static async Task<int> Main()</code>	No se usa <code>args</code> , se usa <code>await</code>
<code>static async Task<int> Main(string[] args)</code>	Se usan <code>args</code> y <code>await</code>

Si el valor devuelto de `Main` no se usa, la devolución de `void` o `Task` permite que el código sea ligeramente más sencillo.

 Expandir tabla

declaración <code>Main</code>	Código del método <code>Main</code>
<code>static void Main()</code>	No se usa <code>args</code> ni <code>await</code>
<code>static void Main(string[] args)</code>	Se usa <code>args</code> , no se usa <code>await</code>
<code>static async Task Main()</code>	No se usa <code>args</code> , se usa <code>await</code>
<code>static async Task Main(string[] args)</code>	Se usan <code>args</code> y <code>await</code>

Pero, si se devuelve `int` o `Task<int>`, el programa puede comunicar información de estado a otros programas o scripts que invocan el archivo ejecutable.

En el ejemplo siguiente se muestra cómo se puede acceder al código de salida para el proceso.

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](#).

Cree una aplicación mediante la ejecución de `dotnet new console`. Modifique el método `Main` en `Program.cs` como se indica a continuación:

```
C#  
  
// 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.

```
PowerShell  
  
dotnet run  
if ($LastExitCode -eq 0) {  
    Write-Host "Execution succeeded"  
} else  
{  
    Write-Host "Execution Failed"  
}  
Write-Host "Return value = " $LastExitCode
```

```
Output
```

```
Execution succeeded  
Return value = 0
```

Valores devueltos asincrónicos de Main

Cuando se declara un valor devuelto `async` para `Main`, el compilador genera el código reutilizable para llamar a métodos asincrónicos en `Main`. Si no especifica la palabra clave `async`, debe escribir ese código usted mismo, como se muestra en el siguiente ejemplo. El código del ejemplo garantiza que el programa se ejecute hasta que se complete la operación asincrónica:

```
C#  
  
class AsyncMainReturnValTest  
{  
    public static int Main()  
    {  
        return AsyncConsoleWork().GetAwaiter().GetResult();  
    }  
  
    private static async Task<int> AsyncConsoleWork()  
    {  
        // Main body here  
        return 0;  
    }  
}
```

Este código reutilizable se puede reemplazar por:

```
C#  
  
class Program  
{  
    static async Task<int> Main(string[] args)  
    {  
        return await AsyncConsoleWork();  
    }  
  
    private static async Task<int> AsyncConsoleWork()  
    {  
        // main body here  
        return 0;  
    }  
}
```

Una ventaja de declarar `Main` como `async` es que el compilador siempre genera el código correcto.

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();`

ⓘ Nota

Si en los ejemplos se usase el modificador `async` en el método `Main`, el compilador generaría el mismo código.

Argumentos de la línea de comandos

Puede enviar argumentos al método `Main` definiéndolo de una de las siguientes maneras:

 Expandir tabla

declaración <code>Main</code>	Código del método <code>Main</code>
<code>static void Main(string[] args)</code>	No hay valores devueltos, no se usa <code>await</code>
<code>static int Main(string[] args)</code>	Valor devuelto, no se usa <code>await</code>
<code>static async Task Main(string[] args)</code>	No hay valores devueltos, se usa <code>await</code>
<code>static async Task<int> Main(string[] args)</code>	Valor devuelto, se usa <code>await</code>

Si no se usan los argumentos, puede omitir `args` de la declaración del método para simplificar ligeramente el código:

 Expandir tabla

declaración Main	Código del método Main
<code>static void Main()</code>	No hay valores devueltos, no se usa <code>await</code>
<code>static int Main()</code>	Valor devuelto, no se usa <code>await</code>
<code>static async Task Main()</code>	No hay valores devueltos, se usa <code>await</code>
<code>static async Task<int> Main()</code>	Valor devuelto, se usa <code>await</code>

Nota

También puede usar [Environment.CommandLine](#) o [Environment.GetCommandLineArgs](#) para acceder a los argumentos de la línea de comandos desde cualquier punto de una consola o de una aplicación de Windows Forms. Para habilitar los argumentos de la línea de comandos en la declaración del método `Main` de una aplicación Windows Forms, tendrá que modificar manualmente la declaración de `Main`. El código generado por el diseñador de Windows Forms crea un `Main` sin ningún parámetro de entrada.

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:

C#

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

Sugerencia

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](#):

C#

```
long num = Int64.Parse(args[0]);
```

También se puede usar el tipo de C# `long`, que tiene como alias `Int64`:

C#

```
long num = long.Parse(args[0]);
```

También puede usar el método `ToInt64` de la clase `Convert` para hacer lo mismo:

C#

```
long num = Convert.ToInt64(s);
```

Para obtener más información, vea [Parse](#) y [Convert](#).

💡 Sugerencia

El análisis de argumentos de línea de comandos puede ser complejo. Considere la posibilidad de usar la biblioteca [SystemCommandLine](#) (actualmente en versión beta) para simplificar el proceso.

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*.

C#

```
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}.");
    }
}
```

```
// 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 creó.

3. Escriba el siguiente comando para compilar la aplicación.

```
dotnet build
```

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:

```
dotnet run -- 3
```

5. El comando genera este resultado: `The factorial of 3 is 6.`

ⓘ Nota

Al ejecutar una aplicación en Visual Studio, puede especificar argumentos de la línea de comandos en la [Página Depuración, Diseñador de proyectos](#).

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.Environment](#)
- [Procedimiento para mostrar argumentos de la línea de comandos](#)



Colaborar con nosotros en
[GitHub](#)

El origen de este contenido se
puede encontrar en GitHub,
donde también puede crear y

.NET

Comentarios de .NET

.NET es un proyecto de código
abierto. Seleccione un vínculo para
proporcionar comentarios:

revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

 Abrir incidencia con la documentación

 Proporcionar comentarios sobre el producto

Instrucciones de nivel superior: programas sin métodos Main

Artículo • 01/03/2024

No es necesario incluir explícitamente un método `Main` en un proyecto de aplicación de consola. En su lugar, puede usar la característica de *instrucciones de nivel superior* para minimizar el código que tiene que escribir.

Las instrucciones de nivel superior permiten escribir código ejecutable directamente en la raíz de un archivo, lo que elimina la necesidad de encapsular el código en una clase o método. Esto significa que puede crear programas sin la ceremonia de una clase `Program` y un método `Main`. En este caso, el compilador genera una clase `Program` con un método de punto de entrada para la aplicación. El nombre de este método no es en realidad `Main`, es un detalle de implementación al que el código no puede hacer referencia directamente.

Aquí se muestra un archivo `Program.cs` que es un programa de C# completo en C# 10:

```
C#  
  
Console.WriteLine("Hello World!");
```

Las instrucciones de nivel superior permiten escribir programas sencillos para utilidades pequeñas, como Azure Functions y Acciones de GitHub. También facilitan a los nuevos programadores de C# empezar a aprender y escribir código.

En las secciones siguientes se explican las reglas de lo que puede y no puede hacer con las instrucciones de nivel superior.

Solo un archivo de nivel superior

Una aplicación solo debe tener un punto de entrada. Un proyecto solo puede tener un archivo con instrucciones de nivel superior. Al colocar instrucciones de nivel superior en más de un archivo de un proyecto, se produce el error del compilador siguiente:

CS8802 Solo una unidad de compilación puede tener instrucciones de nivel superior.

Un proyecto puede tener cualquier número de archivos de código fuente adicionales que no tengan instrucciones de nivel superior.

Ningún otro punto de entrada

Puede escribir un método `Main` de forma explícita, pero no puede funcionar como punto de entrada. El compilador emite la advertencia siguiente:

CS7022 El punto de entrada del programa es código global: se ignora el punto de entrada "Main()".

En un proyecto con instrucciones de nivel superior, no se puede usar la opción del compilador `-main` para seleccionar el punto de entrada, incluso si el proyecto tiene uno o varios métodos `Main`.

Directivas `using`

Si incluye directivas `using`, deben aparecer en primer lugar en el archivo, como en este ejemplo:

```
C#  
  
using System.Text;  
  
StringBuilder builder = new();  
builder.AppendLine("Hello");  
builder.AppendLine("World!");  
  
Console.WriteLine(builder.ToString());
```

Espacio de nombres global

Las instrucciones de nivel superior están implícitamente en el espacio de nombres global.

Espacios de nombres y definiciones de tipos

Un archivo con instrucciones de nivel superior también puede contener espacios de nombres y definiciones de tipos, pero deben aparecer después de las instrucciones de nivel superior. Por ejemplo:

```
C#  
  
MyClass.TestMethod();  
MyNamespace.MyClass.MyMethod();
```

```
public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from
MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

args

Las instrucciones de nivel superior pueden hacer referencia a la variable `args` para acceder a los argumentos de línea de comandos que se hayan escrito. La variable `args` nunca es `NULL`, pero su valor `Length` es cero si no se han proporcionado argumentos de línea de comandos. Por ejemplo:

C#

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

await

Puede llamar a un método asincrónico mediante el uso `await`. Por ejemplo:

C#

```
Console.WriteLine("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

Código de salida para el proceso

Para devolver un valor `int` cuando finaliza la aplicación, use la instrucción `return` como lo haría en un método `Main` que devuelva una instancia de `int`. Por ejemplo:

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

Método de punto de entrada implícito

El compilador genera un método que actúa como el punto de entrada del programa para un proyecto con instrucciones de nivel superior. La firma del método depende de si las instrucciones de nivel superior contienen la palabra clave `await` o la instrucción `return`. En la tabla siguiente se muestra el aspecto que tendría la firma del método, utilizando el nombre del método `Main` en la tabla para mayor comodidad.

[+] Expandir tabla

El código de nivel superior contiene	Signatura de <code>Main</code> implícita
<code>await</code> y <code>return</code>	<code>static async Task<int> Main(string[] args)</code>
<code>await</code>	<code>static async Task Main(string[] args)</code>
<code>return</code>	<code>static int Main(string[] args)</code>
No <code>await</code> ni <code>return</code>	<code>static void Main(string[] args)</code>

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#.

 Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

El sistema de tipos de C#

Artículo • 11/04/2024

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, el tipo y naturaleza (valor, referencia o salida) para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET define tipos numéricos integrados, así como tipos complejos que representan una amplia variedad de construcciones. Entre ellas se incluyen 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 pueden incluir los siguientes elementos:

- 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.
- Interfaces que implementa.
- 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:

C#

```
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;
```

ⓘ Nota

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 utilizar 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:

```
C#  
  
// 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:

```
C#  
  
public string GetName(int ID)  
{  
    if (ID < names.Length)  
        return names[ID];  
    else  
        return String.Empty;  
}  
private string[] names = { "Spencer", "Sally", "Doug" };
```

Después de declarar una variable, no se puede volver a declarar con un nuevo tipo y no se 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 integrados. Estos representan 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. Estos tipos están disponibles para su uso en cualquier programa de C#. Para obtener una lista completa de los tipos integrados, vea [Tipos integrados](#).

Tipos personalizados

Puede usar las construcciones `struct`, `class`, `interface`, `enum` y `record` para crear sus propios tipos personalizados. La biblioteca de clases .NET es en sí misma una colección de tipos personalizados 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 que los define. 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

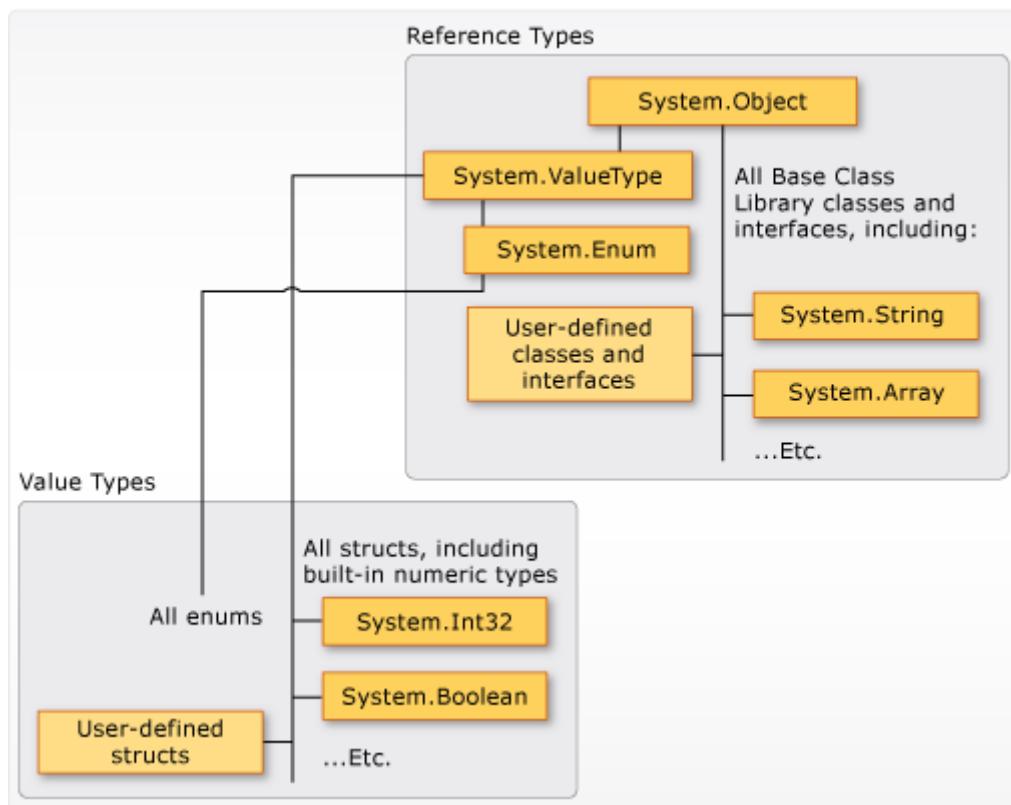
Es importante entender dos aspectos fundamentales sobre el sistema de tipos en .NET:

- 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`), se 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*. Estos tipos incluyen 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` o `record` 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.



ⓘ Nota

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.

Las clases (class) y estructuras (struct) son dos de las construcciones básicas de Common Type System en .NET. C# 9 agrega registros, que son un tipo de clase. 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, estructura o registro. Los miembros incluyen sus métodos, propiedades y eventos, entre otros elementos, como se muestra más adelante en este artículo.

Una declaración de clase, estructura o registro es como un plano que se utiliza para crear instancias u objetos en tiempo de ejecución. Si define una clase, una estructura o un registro denominado `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 del tipo, 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.

Los tipos de registro pueden ser tipos de referencia (`record class`) o tipos de valor (`record struct`).

En general, las clases se utilizan para modelar comportamientos más complejos. Las clases suelen almacenar datos que están diseñados para modificarse después de crear un objeto de clase. Los structs son más adecuados para estructuras de datos pequeñas. Los structs suelen almacenar datos que no están diseñados para modificarse después de que se haya creado el struct. Los tipos de registro son estructuras de datos con miembros sintetizados del compilador adicionales. Los registros suelen almacenar datos que no están diseñados para modificarse después de que se haya creado el objeto.

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. La memoria de un struct se asigna en línea 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. Puede declarar tipos `record` `struct` que son tipos de valor e incluir los miembros sintetizados para los `registros`.

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:

```
C#  
  
// constant field on type byte.  
byte b = byte.MaxValue;
```

Pero se declaran y se les asignan valores como si fueran tipos simples no agregados:

```
C#  
  
byte num = 0xA;  
int i = 5;  
char c = 'Z';
```

Los tipos de valor están *sellados*. No se puede derivar un tipo de cualquier tipo de valor, por ejemplo `System.Int32`. No se 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. No se puede convertir un tipo de struct en cualquier tipo de interfaz que implemente. Esta conversión provoca una operación *boxing* que 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:

```
C#  
  
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:

C#

```
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 `class`, `record`, `delegate`, matriz o `interface` es un `reference type`.

Al declarar una variable de un [reference type](#), contiene el valor `null` hasta que se asigna con una instancia de ese tipo o se crea una mediante el operador `new`. La creación y asignación de una clase se muestran en el ejemplo siguiente:

C#

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

No se puede crear una instancia de [interface](#) directamente mediante el operador `new`. En su lugar, cree y asigne una instancia de una clase que implemente la interfaz. Considere el ejemplo siguiente:

C#

```
MyClass myClass = new MyClass();

// Declare and assign using an existing value.
IMyInterface myInterface = myClass;

// Or create and assign a value in a single statement.
IMyInterface myInterface2 = new MyClass();
```

Cuando se crea un objeto, se asigna memoria en el montón administrado. La variable contiene solo una referencia a la ubicación del objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando se reclaman. La *recolección de elementos no utilizados* es la funcionalidad de administración automática de memoria de CLR, que realiza la recuperación. 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](#). El usuario las declara y las usa con la sintaxis simplificada que proporciona C#, como se muestra en el ejemplo siguiente:

C#

```
// 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 [sellado](#). Otras clases pueden heredar de la clase e invalidar sus métodos virtuales. Para obtener más información sobre cómo crear sus clases, vea [Clases, estructuras y registros](#). 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 `float`, 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:

```
C#  
  
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*). El código de cliente proporciona el tipo concreto cuando crea 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, `string`:

```
C#
```

```
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 con establecimiento inflexible de tipos* 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 `null`. Para más información, vea [Tipos que admiten un valor NULL](#).

Tipo en tiempo de compilación y tipo en tiempo de ejecución

Una variable puede tener distintos tipos en tiempo de compilación y en tiempo de ejecución. El *tipo en tiempo de compilación* es el tipo declarado o inferido de la variable en el código fuente. El *tipo en tiempo de ejecución* es el tipo de la instancia a la que hace referencia esa variable. A menudo, estos dos tipos son los mismos, como en el ejemplo siguiente:

```
C#
```

```
string message = "This is a string of characters";
```

En otros casos, el tipo en tiempo de compilación es diferente, tal y como se muestra en los dos ejemplos siguientes:

```
C#
```

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

En los dos ejemplos anteriores, el tipo en tiempo de ejecución es `string`. El tipo en tiempo de compilación es `object` en la primera línea y `IEnumerable<char>` en la segunda.

Si los dos tipos son diferentes para una variable, es importante comprender cuándo se aplican el tipo en tiempo de compilación y el tipo en tiempo de ejecución. El tipo en tiempo de compilación determina todas las acciones realizadas por el compilador. Estas acciones del compilador incluyen la resolución de llamadas a métodos, la resolución de sobrecarga y las conversiones implícitas y explícitas disponibles. El tipo en tiempo de ejecución determina todas las acciones que se resuelven en tiempo de ejecución. Estas acciones de tiempo de ejecución incluyen el envío de llamadas a métodos virtuales, la evaluación de expresiones `is` y `switch` y otras API de prueba de tipos. Para comprender mejor cómo interactúa el código con los tipos, debe reconocer qué acción se aplica a cada tipo.

Secciones relacionadas

Para más información, consulte los siguientes artículos.

- [Tipos integrados](#)
- [Tipos de valor](#)
- [Tipos de referencia](#)

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#.

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto](#) ↗

Declaración de espacios de nombres para organizar los tipos

Artículo • 13/07/2023

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:

C#

```
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 el nombre completo no sea necesario, como en el ejemplo siguiente:

C#

```
using System;
```

C#

```
Console.WriteLine("Hello World!");
```

Para más información, vea [using \(Directiva\)](#).

ⓘ Importante

Las plantillas de C# para .NET 6 usan *instrucciones de nivel superior*. Es posible que la aplicación no coincida con el código de este artículo si ya ha actualizado a .NET 6. Para obtener más información, consulte el artículo [Las nuevas plantillas de C# generan instrucciones de nivel superior](#).

El SDK de .NET 6 también agrega un conjunto de directivas *implícitas* `global using` para proyectos que usan los SDK siguientes:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Estas directivas de `global using` implícitas incluyen los espacios de nombres más comunes para el tipo de proyecto.

Para saber más, consulte el artículo sobre las [directivas de uso implícito](#)

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:

```
C#  
  
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.

A partir de C# 10, puede declarar un espacio de nombres para todos los tipos definidos en ese archivo, como se muestra en el ejemplo siguiente:

```
C#  
  
namespace SampleNamespace;  
  
class AnotherSampleClass  
{  
    public void AnotherSampleMethod()  
    {  
        System.Console.WriteLine(  
            "SampleMethod inside SampleNamespace");  
    }  
}
```

La ventaja de esta nueva sintaxis es que es más sencilla, lo que ahorra espacio horizontal y llaves. Esto facilita la lectura del código.

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#](#).

Introducción a las clases

Artículo • 15/08/2024

Tipos de referencia

Un tipo que se define como una [class](#), 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 que se le asigne un objeto creado en otro lugar, tal y como se muestra en el ejemplo siguiente:

C#

```
//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. La memoria que un objeto usa la reclama la funcionalidad de administración automática de memoria de CLR, lo que se conoce como *recolección de elementos no utilizados*. 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:

C#

```
//[access modifier] - [class] - [identifier]  
public class Customer  
{  
    // Fields, properties, methods and events go here...  
}
```

Un modificador de acceso opcional precede a la palabra clave `class`. El acceso predeterminado para un tipo `class` es `internal`. 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.

Se pueden crear objetos usando la palabra clave `new` seguida del nombre de la clase, como en este ejemplo:

```
C#
```

```
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:

```
C#
```

```
Customer object2;
```

No se recomienda crear referencias de objeto que no remitan 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. Una referencia puede hacer referencia a un objeto, ya sea creando uno o asignándola a un objeto existente, como en el ejemplo siguiente:

```
C#
```

```
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".

Constructores e inicialización

En las secciones anteriores presentamos la sintaxis para declarar un tipo de clase y crear una instancia de ese tipo. Al crear una instancia de un tipo, conviene asegurarse de que sus campos y propiedades se inicializan con valores útiles. Hay varias maneras de inicializar valores:

- Aceptando los valores predeterminados
- Con inicializadores de campo
- Con parámetros del constructor
- Inicializadores de objeto

Cada tipo de .NET tiene un valor predeterminado, que suele ser 0 en los tipos de número y `null` en todos los tipos de referencia. Puede basarse en ese valor predeterminado si tiene sentido en su aplicación.

Cuando el valor predeterminado de .NET no es el valor correcto, puede establecer un valor inicial usando un *inicializador de campo*:

```
C#  
  
public class Container  
{  
    // Initialize capacity field to a default value of 10:  
    private int _capacity = 10;  
}
```

Para requerir que los autores de llamadas proporcionen un valor inicial, defina un *constructor* que se encargue de establecer ese valor inicial:

```
C#  
  
public class Container  
{  
    private int _capacity;  
  
    public Container(int capacity) => _capacity = capacity;  
}
```

A partir de C# 12, puede definir un *constructor principal* como parte de la declaración de clase:

```
C#  
  
public class Container(int capacity)  
{  
    private int _capacity = capacity;  
}
```

Agregar parámetros al nombre de clase define el *constructor principal*. Esos parámetros están disponibles en el cuerpo de la clase, que incluye sus miembros. Puede usarlos para inicializar campos o en cualquier otro lugar donde los necesite.

También puede usar el modificador `required` en una propiedad y permitir que los autores de llamadas usen un *inicializador de objeto* para establecer el valor inicial de la propiedad:

```
C#  
  
public class Person  
{  
    public required string LastName { get; set; }  
    public required string FirstName { get; set; }  
}
```

Agregar las palabras clave `required` obliga a los llamadores a establecer esas propiedades como parte de una expresión `new`:

```
C#  
  
var p1 = new Person(); // Error! Required properties not set  
var p2 = new Person() { FirstName = "Grace", LastName = "Hopper" };
```

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 clase que no esté definida como `sealed`. Otras clases pueden heredar de la clase e invalidar los métodos virtuales de clase. Además, puede implementar una o varias interfaces.

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:

C#

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Cuando una declaración de clase incluye una clase base, hereda todos los miembros de la clase base, excepto los constructores. Para obtener más información, vea [Herencia](#).

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 una o varias interfaces. Para obtener más información, vea [Interfaces](#).

Una clase se puede declarar como [abstract](#). 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. En cambio, una clase [sellada](#) 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](#).

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#.

Introducción a los tipos de registro en C#

Artículo • 01/06/2023

Un [registro](#) en C# es una [clase](#) o [estructura](#) que proporciona sintaxis y comportamiento especiales para trabajar con modelos de datos. El modificador `record` indica al compilador que sintetice los miembros que son útiles para los tipos cuyo rol principal es almacenar datos. Estos miembros incluyen una sobrecarga de [ToString\(\)](#) y miembros que admiten la igualdad de valores.

Cuándo se usan los registros

Considere la posibilidad de usar un registro en lugar de una clase o estructura en los escenarios siguientes:

- Desea definir un modelo de datos que dependa de la [igualdad de valores](#).
- Desea definir un tipo para el que los objetos son inmutables.

Igualdad de valores

En el caso de los registros, la igualdad de valores significa que dos variables de un tipo de registro son iguales si los tipos coinciden y todos los valores de propiedad y campo coinciden. Para otros tipos de referencia, como las clases, la igualdad significa [igualdad de referencias](#). Es decir, dos variables de un tipo de clase son iguales si hacen referencia al mismo objeto. Los métodos y operadores que determinan la igualdad de dos instancias de registro usan la igualdad de valores.

No todos los modelos de datos funcionan bien con la igualdad de valores. Por ejemplo, [Entity Framework Core](#) depende de la igualdad de referencias para garantizar que solo usa una instancia de un tipo de entidad para lo que es conceptualmente una entidad. Por esta razón, los tipos de registro no son adecuados para su uso como tipos de entidad en Entity Framework Core.

Inmutabilidad

Un tipo inmutable es aquél que impide cambiar cualquier valor de propiedad o campo de un objeto una vez creada su instancia. La inmutabilidad puede ser útil cuando se necesita que un tipo sea seguro para los subprocesos o si depende de que un código

hash quede igual en una tabla hash. Los registros proporcionan una sintaxis concisa para crear y trabajar con tipos inmutables.

La inmutabilidad no es adecuada para todos los escenarios de datos. Por ejemplo, [Entity Framework Core](#) no admite la actualización con tipos de entidad inmutables.

Diferencias entre los registros y las clases y estructuras

La misma sintaxis que [declara](#) y [crea instancias](#) de clases o estructuras se puede usar con los registros. Basta con sustituir la palabra clave `class` por `record`, o bien usar `record struct` en lugar de `struct`. Del mismo modo, las clases de registro admiten la misma sintaxis para expresar las relaciones de herencia. Los registros se diferencian de las clases de las siguientes maneras:

- Puede usar [parámetros posicionales](#) en un [constructor principal](#) para crear un tipo y sus instancias con propiedades inmutables.
- Los mismos métodos y operadores que indican la igualdad o desigualdad de la referencia en las clases (como `Object.Equals(Object)` y `==`), indican la [igualdad o desigualdad de valores](#) en los registros.
- Puede usar una [expresión with](#) para crear una copia de un objeto inmutable con nuevos valores en las propiedades seleccionadas.
- El método `ToString` de un registro crea una cadena con formato que muestra el nombre de tipo de un objeto y los nombres y valores de todas sus propiedades públicas.
- Un registro puede [heredar de otro registro](#). Un registro no puede heredar de una clase y una clase no puede heredar de un registro.

Las estructuras de registro se diferencian de las estructuras en que el compilador sintetiza los métodos para la igualdad y `ToString`. El compilador sintetiza un método `Deconstruct` para las estructuras de registro posicional.

El compilador sintetiza una propiedad pública de solo inicialización de cada parámetro del constructor principal en un objeto `record class`. En `record struct`, el compilador sintetiza una propiedad pública de lectura y escritura. El compilador no crea propiedades para los parámetros del constructor principal en los tipos `class` y `struct` que no incluyan el modificador `record`.

Ejemplos

En el ejemplo siguiente se define un registro público que usa parámetros posicionales para declarar y crear instancias de un registro. A continuación, imprime el nombre de tipo y los valores de propiedad:

```
C#  
  
public record Person(string FirstName, string LastName);  
  
public static void Main()  
{  
    Person person = new("Nancy", "Davolio");  
    Console.WriteLine(person);  
    // output: Person { FirstName = Nancy, LastName = Davolio }  
}
```

En el ejemplo siguiente se muestra la igualdad de valores en los registros:

```
C#  
  
public record Person(string FirstName, string LastName, string[]  
PhoneNumbers);  
  
public static void Main()  
{  
    var phoneNumbers = new string[2];  
    Person person1 = new("Nancy", "Davolio", phoneNumbers);  
    Person person2 = new("Nancy", "Davolio", phoneNumbers);  
    Console.WriteLine(person1 == person2); // output: True  
  
    person1.PhoneNumbers[0] = "555-1234";  
    Console.WriteLine(person1 == person2); // output: True  
  
    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False  
}
```

En el ejemplo siguiente se muestra el uso de una expresión `with` para copiar un objeto inmutable y cambiar una de las propiedades:

```
C#  
  
public record Person(string FirstName, string LastName)  
{  
    public string[] PhoneNumbers { get; init; }  
}  
  
public static void Main()  
{  
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1]  
};  
    Console.WriteLine(person1);
```

```
// output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

Person person2 = person1 with { FirstName = "John" };
Console.WriteLine(person2);
// output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
Console.WriteLine(person1 == person2); // output: False

person2 = person1 with { PhoneNumbers = new string[1] };
Console.WriteLine(person2);
// output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
Console.WriteLine(person1 == person2); // output: False

person2 = person1 with { };
Console.WriteLine(person1 == person2); // output: True
}
```

Para obtener más información, consulte [Registros \(Referencia de C#\)](#).

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#.

Interfaces: definir el comportamiento de varios tipos

Artículo • 18/03/2023

Una interfaz contiene las definiciones de un grupo de funcionalidades relacionadas que una [class](#) o una [struct](#) no abstracta deben implementar. Una interfaz puede definir métodos [static](#), que deben tener una implementación. 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.

C#

```
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 coincide 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. A partir de C# 11, los miembros de interfaz que no son campos pueden ser `static abstract`. Una interfaz no puede contener campos de instancia, constructores de instancias ni finalizadores. Los miembros de interfaz son públicos de forma predeterminada y se pueden especificar explícitamente modificadores de accesibilidad, como `public`, `protected`, `internal`, `private`, `protected internal` o `private protected`. Un miembro `private` debe tener una implementación 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.

ⓘ Nota

Cuando una interfaz declara miembros estáticos, un tipo que implementa esa interfaz también puede declarar miembros estáticos con la misma firma. Los miembros son distintos y se identifican de forma única mediante el tipo que declara el miembro. El miembro estático declarado en un tipo *no* reemplaza el miembro estático que se declara en la interfaz.

Una clase o estructura que implementa una interfaz debe proporcionar una implementación para todos los miembros declarados sin una implementación predeterminada proporcionada por la interfaz. 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`.

C#

```
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 las interfaces declaran una implementación predeterminada de un método, cualquier clase que las implemente heredará la implementación correspondiente (deberá convertir la instancia de clase al tipo de interfaz para acceder a la implementación predeterminada en el miembro de la interfaz).

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:

- En las versiones de C# anteriores a la 8.0, una interfaz es como una clase base abstracta con solo miembros abstractos. Cualquier clase o estructura que

implemente la interfaz debe implementar todos sus miembros.

- A partir de la versión 8.0 de C#, una interfaz puede definir implementaciones predeterminadas para algunos o todos sus miembros. Una clase o estructura que implemente la interfaz no tiene que implementar los miembros que tengan implementaciones predeterminadas. Para obtener más información, vea [Métodos de interfaz predeterminados](#).
- 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.

Clases y métodos genéricos

Artículo • 20/03/2024

Los genéricos presentan el concepto de parámetros de tipo en .NET. Los genéricos permiten diseñar clases y métodos que aplazan la especificación de uno o varios parámetros de tipo hasta que se usa la clase o el método en el código. 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:

C#

```
// 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 parámetros de tipo genérico se reemplazan por los argumentos de tipo durante la compilación. En el ejemplo anterior, el compilador reemplaza `T` por `int`. 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 solo por compatibilidad. Para más información, vea [Elementos genéricos en .NET](#).

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:

- 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.

`T` está disponible para la clase `Node` anidada. 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`.

C#

```
// 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. Si cambia el argumento de tipo, el código siguiente crea listas de cadenas o cualquier otro tipo personalizado:

C#

```

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");
    }
}

```

```
}
```

ⓘ Nota

Los tipos genéricos no se limitan a las clases. Los ejemplos anteriores usan tipos `class`, pero puede definir tipos genéricos `interface` y `struct`, incluidos los tipos `record`.

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`. Las colecciones genéricas 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 en tiempo de ejecución sobre los tipos que se usan en un tipo de datos genérico mediante la reflexión.

Especificación del lenguaje C#

Para obtener más información, consulte la [Especificación del lenguaje C#](#).

Vea también

- [Elementos genéricos en .NET](#)
- [System.Collections.Generic](#)



Colaborar con nosotros en
GitHub



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Tipos anónimos

Artículo • 06/12/2023

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`.

C#

```
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`. El uso de tipos anónimos hace que la consulta devuelva una cantidad de datos menor.

Si no especifica nombres de miembros en el tipo anónimo, el compilador da a los miembros de este tipo el nombre de 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`.

C#

```
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);
}
```

💡 Sugerencia

Puede usar la regla de estilo de .NET [IDE0037](#) para aplicar si se prefieren los nombres de miembros inferidos o explícitos.

También es posible definir un campo por objeto de otro tipo: clase, estructura o incluso otro tipo anónimo. Se realiza mediante el uso de la variable que contiene este objeto igual que en el ejemplo siguiente, donde se crean dos tipos anónimos mediante tipos definidos por el usuario para los que ya se han creado instancias. En ambos casos, el campo `product` del tipo anónimo `shipment` y `shipmentWithBonus` será de tipo `Product` que contiene los valores predeterminados de cada campo. Y el campo `bonus` será de tipo anónimo creado por el compilador.

C#

```
var product = new Product();
var bonus = new { note = "You won!" };
var shipment = new { address = "Nowhere St.", product };
var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };
```

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.

C#

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

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.

Los tipos anónimos admiten la mutación no destructiva en forma de [con expresiones](#). Esto le permite crear una nueva instancia de un tipo anónimo en el que una o varias propiedades tienen nuevos valores:

C#

```
var apple = new { Item = "apples", Price = 1.35 };
var onSale = apple with { Price = 0.79 };
Console.WriteLine(apple);
Console.WriteLine(onSale);
```

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 [object](#) de tipo. Sin embargo, el uso de [object](#) para tipos anónimos anula el propósito de la coincidencia segura. Si tiene que almacenar resultados de consulta o pasarlo 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.

Nota

El nivel de accesibilidad de un tipo anónimo es `internal`; por tanto, dos tipos anónimos definidos en ensamblados diferentes no son del mismo tipo. Por tanto, las instancias de tipos anónimos no pueden ser iguales entre sí cuando se definen en ensamblados diferentes, incluso cuando todas sus propiedades son iguales.

Los tipos anónimos invalidan el método `ToString`, concatenando el nombre y la salida `ToString` de cada propiedad rodeada de llaves.

```
var v = new { Title = "Hello", Age = 24 };

Console.WriteLine(v.ToString()); // "{ Title = Hello, Age = 24 }"
```

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Información general de clases, estructuras y registros en C#

Artículo • 11/04/2024

En C#, la definición de un tipo (una clase, estructura o registro) es como un plano técnico que especifica lo que el tipo puede hacer. Un objeto es básicamente un bloque de memoria que se ha asignado y configurado de acuerdo con el plano. En este artículo se proporciona información general de estos planos técnicos y sus características. En el [siguiente artículo de esta serie](#) se presentan los objetos.

Encapsulación

A veces se hace referencia a la *encapsulación* como el primer pilar o principio de la programación orientada a objetos. 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, consulte el tutorial [Programación orientada a objetos](#).

Miembros

Los *miembros* de un tipo incluyen todos los métodos, campos, constantes, propiedades y eventos. En C#, no hay métodos ni variables globales como en otros lenguajes. Incluso se debe declarar el punto de entrada de un programa, el método `Main`, dentro de una clase o estructura (de forma implícita cuando usa [instrucciones de nivel superior](#)).

La lista siguiente incluye los diversos tipos de miembros que se pueden declarar en una clase, estructura o registro.

- Campos
- Constantes
- Propiedades
- Métodos
- Constructores
- Events
- Finalizadores
- Indexadores
- Operadores

- Tipos anidados

Para más información, consulte [Miembros](#).

Accesibilidad

Algunos métodos y propiedades están diseñados para ser invocables y accesibles desde el código fuera de una clase o 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 siguientes modificadores de acceso para especificar hasta qué punto los tipos y sus miembros son accesibles para el código de cliente:

- [public](#)
- [protected](#)
- [internal](#)
- [protected internal](#)
- [private](#)
- [private protected](#)

La accesibilidad predeterminada es [private](#).

Herencia

Las clases (pero no las estructuras) admiten el concepto de herencia. Una clase que deriva de otra clase (denominada *clase base*) contiene automáticamente todos los miembros públicos, protegidos e internos de la clase base, salvo sus constructores y finalizadores.

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 [Herencia y Polimorfismo](#).

Interfaces

Las clases, las estructuras y los registros pueden implementar varias interfaces. Implementar de una interfaz significa que el tipo implementa todos los métodos definidos en la interfaz. Para más información, vea [Interfaces](#).

Tipos genéricos

Las clases, las estructuras y los registros 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 ni los registros) 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, las estructuras y los registros pueden contener miembros estáticos. Para obtener más información, vea [Clases estáticas y sus miembros](#).

Tipos anidados

Una clase, estructura o registro se puede anidar dentro de otra clase, estructura o registro. 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 e inicializar objetos de clase o estructura, así como colecciones de objetos, asignando valores a sus propiedades. Para más información, consulte [Procedimiento para inicializar un objeto mediante un inicializador de objeto](#).

Tipos anónimos

En situaciones donde no es conveniente o necesario crear una clase con nombre, utilice los tipos anónimos. Los tipos anónimos se definen mediante sus miembros de datos con nombre. Para obtener más información, consulte [Tipos anónimos \(Guía de programación de C#\)](#).

Métodos de extensión

Puede "extender" una clase sin crear una clase derivada mediante la creación de un tipo independiente. Ese tipo contiene métodos a los que se puede llamar como si perteneciesen al tipo original. Para más información, consulte [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 de una variable en tiempo de compilación. Para más información, consulte [var \(referencia de C#\)](#).

Registros

C# 9 presenta el tipo `record`, un tipo de referencia que se puede crear en lugar de una clase o una estructura. Los registros son clases con un comportamiento integrado para encapsular datos en tipos inmutables. C# 10 presenta el tipo de valor `record struct`. Un registro (`record class` o `record struct`) proporciona las siguientes características:

- Sintaxis concisa para crear un tipo de referencia con propiedades inmutables.
- Igualdad de valores. Dos variables de un tipo de registro son iguales si tienen el mismo tipo y si, por cada campo, los valores en ambos registros son iguales. Las clases usan la igualdad de referencia: dos variables de un tipo de clase son iguales si hacen referencia al mismo objeto.
- Sintaxis concisa para la mutación no destructiva. Una expresión `with` permite crear una copia de una instancia de registro existente, pero con los valores de propiedad especificados modificados.
- Formato integrado para la presentación. El método `ToString` imprime el nombre del tipo de registro y los nombres y valores de las propiedades públicas.
- Compatibilidad con jerarquías de herencia en clases de registro. Las clases de registro admiten la herencia. Las clases de structs no admiten la herencia.

Para obtener más información, consulte [Registros](#).

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#.

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto](#) ↗

Objetos: creación de instancias de tipos

Artículo • 10/05/2023

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.

ⓘ Nota

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 una segunda variable del mismo tipo a la primera variable, ambas variables hacen referencia al objeto de esa dirección. Este punto se analiza con más detalle más adelante en este artículo.

Las instancias de clases se crean mediante el [operador new](#). En el ejemplo siguiente, `Person` es el tipo, y `person1` y `person2` son instancias u objetos de ese tipo.

C#

```
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);
    }
}

/*
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:

C#

```

namespace Example;

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);
}
/*
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 subprocessos. 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](#).

Nota

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 `Object.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:

C#

```
// 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 = new Person("", 42);
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.
```

En algunos casos, la implementación de `System.ValueType` de `Equals` utiliza la conversión boxing y la reflexión. Para obtener información sobre cómo proporcionar un algoritmo de igualdad eficaz que sea específico del tipo, consulte [Definición de la igualdad de valores para un tipo](#). Los registros son tipos de referencia que usan semántica de valores para la igualdad.

- 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](#)
- [object](#)
- [Herencia](#)
- [class](#)
- [Tipos de estructura](#)
- [new \(operador\)](#)
- [Sistema de tipos comunes](#)

Herencia: deriva tipos para crear un comportamiento más especializado

Artículo • 07/04/2023

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`.

ⓘ Nota

Los structs no admiten la herencia, pero pueden implementar interfaces.

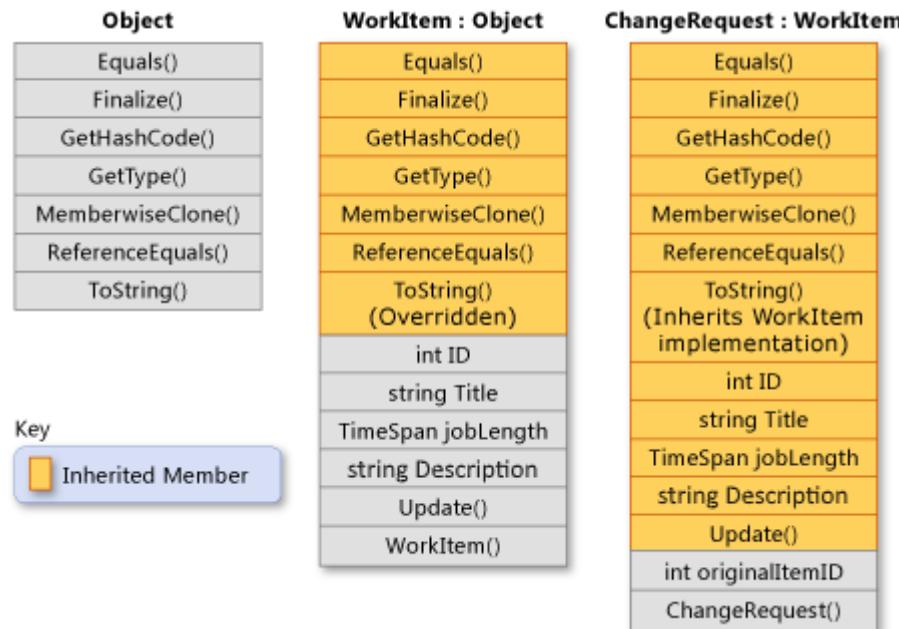
Conceptualmente, una clase derivada es una especialización de la clase base. Por ejemplo, si tiene una clase base `Animal`, podría tener una clase derivada denominada `Mammal` y otra clase derivada denominada `Reptile`. `Mammal` es `Animal` y `Reptile` también es `Animal`, pero cada clase derivada representa 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 más información sobre los métodos de interfaz predeterminados, consulte el artículo sobre [interfaces](#).

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 seis 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:

C#

```
// 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:

C#

```

// 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.

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](#).

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](#).

Polimorfismo

Artículo • 07/04/2023

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 de formas en concreto 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 invalídalo 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`.

C#

```
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).

C#

```

// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever 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, incluye todos los miembros de la clase base. Todo el comportamiento declarado en la clase base forma parte de la clase derivada. Esto permite que los objetos de la clase derivada se traten como objetos de la clase base. Los modificadores de acceso (`public`, `protected`, `private`, etc.) determinan si esos miembros son accesibles desde la implementación de la clase derivada. Los métodos virtuales proporcionan al diseñador diferentes opciones para el comportamiento de la clase derivada:

- 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:

```
C#  
  
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:

```
C#  
  
DerivedClass B = new DerivedClass();  
B.DoWork(); // Calls the new method.  
  
BaseClass A = 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.

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:

```
C#  
  
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:

```
C#  
  
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:

C#

```
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:

C#

```
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:

C#

```
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 enrutaría 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:

C#

```
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](#).

ⓘ Nota

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.

Información general sobre la coincidencia de patrones

Artículo • 13/03/2024

La *coincidencia de patrones* es una técnica consistente en probar una expresión para determinar si tiene ciertas características. La coincidencia de patrones de C# proporciona una sintaxis más concisa para probar las expresiones y realizar acciones cuando una expresión coincide. La "expresión `is`" admite la coincidencia de patrones para probar una expresión y declarar condicionalmente una nueva variable como el resultado de esa expresión. La expresión `switch` permite realizar acciones basadas en el primer patrón que coincide para una expresión. Estas dos expresiones admiten un variado vocabulario de [patrones](#).

En este artículo encontrará información general sobre los escenarios en los que puede usar la coincidencia de patrones. Estas técnicas pueden mejorar la legibilidad y la corrección del código. Para ver una explicación detallada de todos los patrones que puede aplicar, consulte el artículo sobre [patrones](#) en la referencia del lenguaje.

Comprobaciones de valores null

Uno de los escenarios más comunes para usar la coincidencia de patrones es asegurarse de que los valores no son `null`. Puede probar y convertir un tipo de valor que admite valores null a su tipo subyacente mientras prueba `null` con el ejemplo siguiente:

```
C#  
  
int? maybe = 12;  
  
if (maybe is int number)  
{  
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");  
}  
else  
{  
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");  
}
```

El código anterior es un [patrón de declaración](#) para probar el tipo de la variable y asignarlo a una nueva variable. Las reglas de lenguaje hacen que esta técnica sea más segura que muchas otras. Solo es posible acceder a la variable `number` y asignarla en la parte verdadera de la cláusula `if`. Si intenta acceder a ella en otro lugar, ya sea en la

cláusula `else` o después del bloque `if`, el compilador emitirá un error. En segundo lugar, como no usa el operador `==`, este patrón funciona cuando un tipo sobrecarga el operador `==`. Esto lo convierte en una manera ideal de comprobar los valores de referencia nula, incorporando el patrón `not`:

```
C#  
  
string? message = ReadMessageOrDefault();  
  
if (message is not null)  
{  
    Console.WriteLine(message);  
}
```

En el ejemplo anterior se ha usado un *patrón constante* para comparar la variable con `null`. `not` es un *patrón lógico* que coincide cuando el patrón negado no coincide.

Pruebas de tipo

Otro uso común de la coincidencia de patrones consiste en probar una variable para ver si coincide con un tipo determinado. Por ejemplo, el código siguiente comprueba si una variable no es `null` e implementa la interfaz `System.Collections.Generic.IList<T>`. Si es así, usa la propiedad `ICollection<T>.Count` de esa lista para buscar el índice central. El patrón de declaración no coincide con un valor `null`, independientemente del tipo de tiempo de compilación de la variable. El código siguiente protege contra `null`, además de proteger contra un tipo que no implementa `IList`.

```
C#  
  
public static T MidPoint<T>(IEnumerable<T> sequence)  
{  
    if (sequence is IList<T> list)  
    {  
        return list[list.Count / 2];  
    }  
    else if (sequence is null)  
    {  
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be  
null.");  
    }  
    else  
    {  
        int halfLength = sequence.Count() / 2 - 1;  
        if (halfLength < 0) halfLength = 0;  
        return sequence.Skip(halfLength).First();  
    }  
}
```

```
    }  
}
```

Se pueden aplicar las mismas pruebas en una expresión `switch` para probar una variable con varios tipos diferentes. Puede usar esa información para crear algoritmos mejores basados en el tipo de tiempo de ejecución específico.

Comparación de valores discretos

También puede probar una variable para buscar una coincidencia con valores específicos. En el código siguiente se muestra un ejemplo en el que se prueba un valor con todos los valores posibles declarados en una enumeración:

C#

```
public State PerformOperation(Operation command) =>  
    command switch  
    {  
        Operation.SystemTest => RunDiagnostics(),  
        Operation.Start => StartSystem(),  
        Operation.Stop => StopSystem(),  
        Operation.Reset => ResetToReady(),  
        _ => throw new ArgumentException("Invalid enum value for command",  
            nameof(command)),  
    };
```

En el ejemplo anterior se muestra un envío de método basado en el valor de una enumeración. El caso final `_` es un *patrón de descarte* que coincide con todos los valores. Controla todas las condiciones de error en las que el valor no coincide con uno de los valores `enum` definidos. Si omite ese segmento modificador, el compilador le advierte de que la expresión de patrón no controla todos los valores de entrada posibles. En tiempo de ejecución, la expresión `switch` produce una excepción si el objeto que se está examinando no coincide con ninguno de los segmentos modificadores. Puede usar constantes numéricas en lugar de un conjunto de valores de enumeración. También puede usar esta técnica similar para los valores de cadena constantes que representan los comandos:

C#

```
public State PerformOperation(string command) =>  
    command switch  
    {  
        "SystemTest" => RunDiagnostics(),  
        "Start" => StartSystem(),  
        "Stop" => StopSystem(),
```

```
    "Reset" => ResetToReady(),
    _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
};
```

En el ejemplo anterior se muestra el mismo algoritmo, pero se usan valores de cadena en lugar de una enumeración. Usaría este escenario si la aplicación responde a comandos de texto, en lugar de a un formato de datos normal. A partir de C# 11, también puede usar `Span<char>` o `ReadOnlySpan<char>` para probar los valores de cadena constantes, tal como se muestra en el ejemplo siguiente:

C#

```
public State PerformOperation(ReadOnlySpan<char> command) =>
    command switch
    {
        "SystemTest" => RunDiagnostics(),
        "Start" => StartSystem(),
        "Stop" => StopSystem(),
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
    };
```

En todos estos ejemplos, el *patrón de descarte* le garantiza que controlará todas las entradas. El compilador le ayuda a asegurarse de que se controlan todos los valores de entrada posibles.

Patrones relacionales

Puede usar *patrones relacionales* para probar cómo se compara un valor con las constantes. Por ejemplo, el código siguiente devuelve el estado del agua en función de la temperatura en Fahrenheit:

C#

```
string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        (> 32) and (< 212) => "liquid",
        < 32 => "solid",
        > 212 => "gas",
        32 => "solid/liquid transition",
        212 => "liquid / gas transition",
    };
```

El código anterior también muestra el `and` *patrón lógico* conjuntivo para comprobar que ambos patrones relacionales coincidan. También puede usar un patrón `or` disyuntivo para comprobar que cualquiera de los patrones coincide. Los dos patrones relacionales están entre paréntesis, que se pueden usar alrededor de cualquier patrón para mayor claridad. Los dos últimos segmentos modificadores controlan los casos del punto de fusión y el punto de ebullición. Sin esos dos segmentos, el compilador le advertirá de que la lógica no abarca todas las entradas posibles.

El código anterior también muestra otra característica importante que el compilador proporciona para las expresiones de coincidencia de patrones: el compilador le advierte si no controla todos los valores de entrada. El compilador emite además una advertencia si el patrón de una parte de la expresión switch está controlado por un patrón anterior. Esto le da libertad para refactorizar y reordenar expresiones switch. La misma expresión también se podría escribir así:

```
C#  
  
string WaterState2(int tempInFahrenheit) =>  
    tempInFahrenheit switch  
    {  
        < 32 => "solid",  
        32 => "solid/liquid transition",  
        < 212 => "liquid",  
        212 => "liquid / gas transition",  
        _ => "gas",  
    };
```

La lección más importante del ejemplo anterior, y cualquier otra refactorización o reordenación, es que el compilador valida que el código controla todas las entradas posibles.

Varias entradas

Todos los patrones cubiertos hasta ahora comprueban una entrada. Puede escribir patrones que examinen varias propiedades de un objeto. Fíjese en el siguiente registro

`Order`:

```
C#  
  
public record Order(int Items, decimal Cost);
```

El tipo de registro posicional anterior declara dos miembros en posiciones explícitas. Primero aparece `Items` y, luego, el valor `Cost` del pedido. Para obtener más

información, consulte [Registros](#).

El código siguiente examina el número de elementos y el valor de un pedido para calcular un precio con descuento:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
}
```

Los dos primeros segmentos examinan dos propiedades de `Order`. El tercero examina solo el costo. El siguiente realiza la comprobación de `null` y el último coincide con cualquier otro valor. Si el tipo `Order` define un método `Deconstruct` adecuado, puede omitir los nombres de propiedad del patrón y usar la desconstrucción para examinar las propiedades:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10, > 1000.00m ) => 0.10m,
        ( > 5, > 50.00m ) => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
}
```

El código anterior muestra el [patrón posicional](#) donde las propiedades se destruyen para la expresión.

Patrones de lista

Puede comprobar los elementos de una lista o una matriz mediante un *patrón de lista*. Un [patrón de lista](#) proporciona una forma de aplicar un patrón a cualquier elemento de una secuencia. Además, puede aplicar el *patrón de descarte* (`_`) para que coincida con

cualquier elemento, o bien un *patrón de sector* para que coincida con ninguno o más elementos.

Los patrones de lista son una herramienta valiosa cuando los datos no siguen una estructura normal. Puede usar la coincidencia de patrones para probar la forma y los valores de los datos en vez de transformarlos en un conjunto de objetos.

Tenga en cuenta el siguiente extracto de un archivo de texto que contiene transacciones bancarias:

Output			
04-01-2020,	DEPOSIT,	Initial deposit,	2250.00
04-15-2020,	DEPOSIT,	Refund,	125.65
04-18-2020,	DEPOSIT,	Paycheck,	825.65
04-22-2020,	WITHDRAWAL,	Debit, Groceries,	255.73
05-01-2020,	WITHDRAWAL,	#1102, Rent, apt,	2100.00
05-02-2020,	INTEREST,		0.65
05-07-2020,	WITHDRAWAL,	Debit, Movies,	12.57
04-15-2020,	FEE,		5.55

Es un formato CSV, pero algunas de las filas tienen más columnas que otras. También hay algo incluso peor para el procesamiento, una columna del tipo `WITHDRAWAL` que contiene texto generado por el usuario y puede contener una coma en el texto. Un *patrón de lista* que incluye el patrón de *descarte*, el patrón *constante* y el patrón *var* para capturar los datos de los procesos del valor en este formato:

```
C#  
  
decimal balance = 0m;  
foreach (string[] transaction in ReadRecords())  
{  
    balance += transaction switch  
    {  
        [_, "DEPOSIT", _, var amount]      => decimal.Parse(amount),  
        [_, "WITHDRAWAL", ..., var amount] => -decimal.Parse(amount),  
        [_, "INTEREST", var amount]       => decimal.Parse(amount),  
        [_, "FEE", var fee]                => -decimal.Parse(fee),  
        _                                => throw new  
            InvalidOperationException($"Record {string.Join(", ", transaction)} is not  
            in the expected format!"),  
    };  
    Console.WriteLine($"Record: {string.Join(", ", transaction)}, New  
balance: {balance:C}");  
}
```

En el ejemplo anterior se toma una matriz de cadenas, donde cada elemento es un campo de la fila. Las claves de expresión `switch` del segundo campo, que determinan el

tipo de transacción y el número de columnas restantes. Cada fila garantiza que los datos están en el formato correcto. El patrón de descarte (`_`) omite el primer campo, con la fecha de la transacción. El segundo campo coincide con el tipo de transacción. Las coincidencias restantes del elemento saltan hasta el campo con la cantidad. La coincidencia final usa el patrón `var` para capturar la representación de cadena de la cantidad. La expresión calcula la cantidad que se va a agregar o restar del saldo.

Los *patrones de lista* permiten buscar coincidencias en la forma de una secuencia de elementos de datos. Los patrones de *descarte* y de *sector* se usan para hallar coincidencias con la ubicación de los elementos. Se usan otros patrones para que coincidan con las características de los elementos individuales.

En este artículo se proporciona una introducción a los tipos de código que se pueden escribir con la coincidencia de patrones en C#. En los artículos siguientes se muestran más ejemplos del uso de patrones en escenarios y un vocabulario completo de patrones disponibles.

Vea también

- [Uso de la coincidencia de patrones para evitar la comprobación de "is" seguida de una conversión \(reglas de estilo IDE0020 e IDE0038\)](#)
- [Exploración: Uso de la coincidencia de patrones para crear el comportamiento de la clase y mejorar el código](#)
- [Tutorial: Uso de la coincidencia de patrones para compilar algoritmos basados en tipos y basados en datos](#)
- [Referencia: Coincidencia de patrones](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Descartes: aspectos básicos de C#

Artículo • 14/11/2023

Los descartes son variables de marcador de posición que deliberadamente no se usan en el código de la aplicación. Los descartes son equivalentes a variables sin asignar, ya que no tienen un valor. Un descarte comunica la intención al compilador y otros usuarios que leen el código: Pretendía ignorar el resultado de una expresión. Es posible que desee ignorar el resultado de una expresión, uno o varios miembros de una expresión de tupla, un parámetro `out` de un método o el destino de una expresión de coincidencia de patrones.

Los descartes aclaran la intención del código. Un descarte indica que el código nunca usa la variable. Mejoran la legibilidad y el mantenimiento.

Para indicar que una variable es un descarte, se le asigna como nombre el carácter de subrayado (`_`). Por ejemplo, la siguiente llamada de método devuelve una tupla en la que el primer y el segundo valor son descartes. `area` es una variable declarada previamente establecida en el tercer componente devuelto por `GetCityInformation`:

```
C#  
(_, _, area) = city.GetCityInformation(cityName);
```

Puede usar descartes para especificar parámetros de entrada sin usar de una expresión lambda. Para más información, consulte sección sobre [parámetros de entrada de una expresión lambda](#) en el artículo sobre [expresiones lambda](#).

Cuando `_` es un descarte válido, si se intenta recuperar su valor o usarlo en una operación de asignación, se genera el error del compilador CS0103: "El nombre '`_`' no existe en el contexto actual". Este error 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 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 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.

C#

```
var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

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.

C#

```
using System;

namespace Discards
{
    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;
        }
    }
    class Example
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fName, _, city, _) = p;
```

```
        Console.WriteLine($"Hello {fName} of {city}!");
        // The example displays the following output:
        //      Hello John of Boston!
    }
}
}
```

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`

El *patrón de descarte* se puede usar en la coincidencia de patrones con la expresión `switch`. Todas las expresiones, incluida `null`, siempre coinciden con el patrón de descarte.

En el ejemplo siguiente se define un método `ProvidesFormatInfo` que usa una expresión `switch` 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.

C#

```
object?[] objects = [CultureInfo.CurrentCulture,
                      CultureInfo.CurrentCulture.DateTimeFormat,
                      CultureInfo.CurrentCulture.NumberFormat,
                      new ArgumentException(), null];
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object? obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a
NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
//      System.Globalization.CultureInfo 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 a métodos con parámetros `out`

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.

C#

```
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. Un uso típico es usar una asignación para asegurarse de que un argumento no sea `NULL`. En el código siguiente se usa un descarte para forzar una asignación. El lado derecho de la asignación utiliza el [operador de uso combinado de `NULL`](#) para producir `System.ArgumentNullException` cuando el argumento es `null`. El código no necesita el resultado de la asignación, por lo que se descarta. La expresión fuerza una

comprobación nula. El descarte aclara su intención: el resultado de la asignación no es necesario ni se usa.

C#

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg),
message: "arg can't be null");

    // Do work with arg.
}
```

En el ejemplo siguiente se usa un descarte independiente para omitir el objeto `Task` devuelto por una operación asincrónica. La asignación de la tarea tiene el efecto de suprimir la excepción que se produce en la operación cuando está a punto de completarse. Hace que su intención sea clara: Quiere descartar `Task` y omitir los errores generados a partir de esa operación asincrónica.

C#

```
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
```

Sin asignar la tarea a un descarte, el código siguiente genera una advertencia del compilador:

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current
```

```

method continues before the call is completed.
// Consider applying the 'await' operator to the result of the call.
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");

```

① Nota

Si ejecuta cualquiera de los dos ejemplos anteriores mediante un depurador, este detendrá el programa cuando se produzca la excepción. Sin un depurador asociado, la excepción se omite en ambos casos en modo silencioso.

`_` 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:

```

C#

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:

```

C#

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:

C#

```
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
```

Consulte también

- Eliminación de valores de expresión innecesarios (regla de estilo IDE0058)
- Eliminación de la asignación de valores innecesarios (regla de estilo IDE0059)
- Eliminación del parámetro sin usar (regla de estilo IDE0060)
- Deconstruir tuplas y otros tipos
- is operator
- Expresión switch



Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte nuestra guía para colaboradores.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Abrir incidencia con la documentación

Proporcionar comentarios sobre el producto

Deconstruir tuplas y otros tipos

Artículo • 07/04/2023

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. Trabajar elemento a elemento es complicado, como se muestra en el ejemplo siguiente. El método `QueryCityData` devuelve una tupla de tres y cada uno de sus elementos se asigna a una variable en una operación aparte.

C#

```
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);
    }
}
```

Recuperar varios valores de campo y propiedad de un objeto puede ser igualmente complicado: debe asignar un valor de campo o propiedad a una variable miembro a miembro.

Puede recuperar varios elementos de una tupla o recuperar varios campos, propiedades y valores calculados de un objeto en una sola operación de *deconstrucción*. Para deconstruir una tupla, asigne sus elementos a variables individuales. Cuando se deconstruye un objeto, los valores seleccionados se asignan a variables individuales.

Tuplas

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 siguiente instrucción asigna los elementos de una tupla de cuatro a cuatro variables distintas:

```
C#
```

```
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 tres que devuelve el método `QueryCityData`.

```
C#
```

```
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 tres devuelta por el método `QueryCityData`.

```
C#
```

```
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.

```
C#
```

```

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.

C#

```

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.
}

```

- A partir de C# 10, puede mezclar la declaración y la asignación de variables en una deconstrucción.

C#

```

public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}

```

No se puede especificar un tipo específico fuera de los paréntesis aunque cada campo de la tupla tenga el mismo tipo. Al hacerlo, se genera el error del compilador CS8136: "La forma de deconstrucción "var (...)" no permite un tipo específico para "var"".

Debe asignar cada elemento de la tupla a una variable. Si omite algún elemento, el compilador genera el error CS8132: "No se puede deconstruir una tupla de "x" elementos en "y" variables".

Elementos de tupla con descartes

A menudo, cuando se deconstruye una tupla, solo interesan los valores de algunos elementos. Puede aprovechar la compatibilidad de C# con los *descartes*, que son variables de solo escritura cuyos valores se decide omitir. Un carácter de subrayado ("_") elige un descarte 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 seis con el nombre de una ciudad, su área, 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.

C#

```
using System;

public class ExampleDiscard
{
    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)
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
```

Tipos definidos por el usuario

C# no ofrece compatibilidad integrada con la deconstrucción de tipos que no son de tupla distintos de los tipos `record` y `DictionaryEntry`. A pesar de ello, como autor de una clase, una estructura o una interfaz, puede permitir que las instancias del tipo se destruyan 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:

C#

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

A continuación, puede deconstruir una instancia de la clase `Person` denominada `p` con una asignación como el código siguiente:

C#

```
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, el segundo nombre y los apellidos.
- El nombre de pila, los apellidos, el nombre de la ciudad y el nombre del estado.

C#

```
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 ExampleClassDeconstruction
{
    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!
```

Varios métodos de `Deconstruct` que tienen el mismo número de parámetros son ambiguos. Debe tener cuidado al definir métodos `Deconstruct` con distintos números de parámetros o "aridad". No se pueden distinguir los métodos `Deconstruct` con el mismo número de parámetros durante la resolución de sobrecarga.

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.

C#

```
// Deconstruct the person object.  
var ( fName, _, city, _ ) = p;  
Console.WriteLine($"Hello {fName} of {city}!");  
// The example displays the following output:  
//      Hello John of Boston!
```

Métodos de extensión para tipos definidos por el usuario

Aunque usted no haya creado una clase, una estructura o una interfaz, 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 tanto el descriptor de acceso get como set tienen la misma accesibilidad, la variable `access`

indica la accesibilidad de la propiedad en su conjunto. En caso contrario, la accesibilidad de los descriptores de acceso get y set se indica mediante las variables `getAccess` y `setAccess`.

C#

```
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 ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\\n{propType.Name}");
        Console.WriteLine($"  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.WriteLine($"\\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

```

Método de extensión para tipos de sistema

Algunos tipos de sistema proporcionan el método `Deconstruct` por motivos prácticos.

Por ejemplo, el tipo `System.Collections.Generic.KeyValuePair<TKey,TValue>` proporciona esta funcionalidad. Al recorrer en iteración un objeto

`System.Collections.Generic.Dictionary<TKey,TValue>`, cada elemento es un valor `KeyValuePair<TKey, TValue>` y se puede deconstruir. Considere el ejemplo siguiente:

C#

```

Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
}

```

```
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November
10th, 2021.");
}
```

Puede agregar un método `Deconstruct` a tipos de sistema que no tengan uno. Considere el siguiente método de extensión:

C#

```
public static class NullableExtensions
{
    public static void Deconstruct<T>(
        this T? nullable,
        out bool hasValue,
        out T value) where T : struct
    {
        hasValue = nullable.HasValue;
        value = nullable.GetValueOrDefault();
    }
}
```

Este método de extensión permite que todos los tipos `Nullable<T>` se deconstruyan en una tupla de `(bool hasValue, T value)`. En el ejemplo siguiente se muestra el código que usa este método de extensión:

C#

```
DateTime? questionableDateTime = default;
var (hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

questionableDateTime = DateTime.Now;
(hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

// Example outputs:
// { HasValue = False, Value = 1/1/0001 12:00:00 AM }
// { HasValue = True, Value = 11/10/2021 6:11:45 PM }
```

Tipos record

Cuando se declara un tipo de [registro](#) con dos o más parámetros posicionales, el compilador crea un método `Deconstruct` con un parámetro `out` para cada parámetro posicional en la declaración de `record`. Para obtener más información, vea [Sintaxis posicional para la definición de propiedades](#) y [Comportamiento de deconstructores en registros derivados](#).

Vea también

- [Deconstrucción de declaraciones de variables \(regla de estilo IDE0042\)](#)
- [Descartes](#)
- [Tipos de tupla](#)

Excepciones y control de excepciones

Artículo • 11/04/2024

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 se genera una excepción, CLR desenreda 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.

```
C#  
  
public class ExceptionTest  
{  
    static double SafeDivision(double x, double y)  
    {  
        if (y == 0)  
            throw new DivideByZeroException();  
        return x / y;  
    }  
  
    public static void Main()  
    {  
        // Input for test purposes. Change the values to see  
        // exception handling behavior.  
        double a = 98, b = 0;  
        double result;  
  
        try  
        {  
            result = SafeDivision(a, b);  
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);  
        }  
    }  
}
```

```
        catch (DivideByZeroException)
    {
        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).

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](#)
 - [Palabras clave de C#](#)
 - [throw](#)
 - [try-catch](#)
 - [try-finally](#)
 - [try-catch-finally](#)
 - [Excepciones](#)
-

Comentarios

¿Le ha resultado útil esta página?



[Proporcionar comentarios sobre el producto](#) ↗

Uso de excepciones

Artículo • 10/05/2023

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:

```
C#  
  
class CustomException : Exception  
{  
    public CustomException(string message)  
    {  
    }  
}  
private static void TestThrow()  
{  
    throw new CustomException("Custom exception in TestThrow()");  
}
```

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:

```
C#  
  
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 elemento `catch` coincidente, el entorno de ejecución 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. Ordene los bloques `catch` de más específicos (o más derivados) a menos específicos. Por ejemplo:

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (Exception 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 en el entorno de ejecución finalice los objetos. Por ejemplo:

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
        // IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        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 subprocesso, este finaliza.

Control de excepciones (Guía de programación de C#)

Artículo • 20/03/2023

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. Un bloque `finally` contiene 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`.

C#

```
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.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
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 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 distintas clases 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 una clase de excepciones coincidente, se selecciona un bloque `catch` que no tenga ningún tipo, si hay alguno en la instrucción. Es importante colocar primero los bloques `catch` con las clases de excepción más específicas (es decir, las más derivadas).

Detecte excepciones cuando se cumplan las condiciones siguientes:

- 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.

```
C#
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
```

```

    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}

```

- Quiere controlar parcialmente una excepción antes de pasarla para aumentar su control. 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.

C#

```

try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}

```

También puede especificar *filtros de excepción* para agregar una expresión booleana a una cláusula `catch`. Los filtros de excepción indican que una cláusula `catch` específica solo coincide cuando la condición es "true". En el ejemplo siguiente, ambas cláusulas `catch` usan la misma clase de excepción, pero se comprueba una condición adicional para crear un mensaje de error distinto:

C#

```

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}

```

Se puede usar un filtro de excepción que siempre devuelva `false` para examinar todas las excepciones, pero no para procesarlas. Un uso habitual es registrar excepciones:

```
C#  
  
public static void Main()  
{  
    try  
    {  
        string? s = null;  
        Console.WriteLine(s.Length);  
    }  
    catch (Exception e) when (LogException(e))  
    {  
    }  
    Console.WriteLine("Exception must have been handled");  
}  
  
private static bool LogException(Exception e)  
{  
    Console.WriteLine($"\\tIn the log routine. Caught {e.GetType()}");  
    Console.WriteLine($"\\tMessage: {e.Message}");  
    return false;  
}
```

El método `LogException` siempre devuelve `false`; ninguna cláusula `catch` que use este filtro de excepción coincide. La cláusula `catch` puede ser general, mediante el uso de `System.Exception`, y las cláusulas posteriores pueden procesar clases de excepción más específicas.

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. Un bloque `finally` siempre se ejecuta, 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.

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 manipulador de

archivos sigue teniendo el valor `null`, y el bloque `finally` no intenta cerrarlo. En lugar de eso, si el archivo se abre correctamente en el bloque `try`, el bloque `finally` cierra el archivo abierto.

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    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#.

Consulte también

- [Referencia de C#](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [using \(Instrucción\)](#)

Creación e inicio de excepciones

Artículo • 23/12/2023

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, a continuación, se *inician* con la [instrucción o expresión](#). 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 función definida. Por ejemplo, si un parámetro de un método tiene un valor no válido:

C#

```
static void CopyObject(SampleClass original)
{
    _ = original ?? throw new ArgumentException("Parameter cannot be
null", nameof(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 permite una operación, genere una instancia de [InvalidOperationException](#) o un objeto con base en una derivación de esta clase. El código siguiente es un ejemplo de un método que genera un objeto [InvalidOperationException](#):

C#

```
public class ProgramLog
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new 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](#):

```
C#
```

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "Parameter index is out of range.", e);
    }
}
```

⚠ Nota

En el ejemplo anterior se muestra cómo usar la propiedad [InnerException](#). Está simplificada intencionadamente. En la práctica, debe comprobar que un índice está en el intervalo antes de usarlo. Puede usar esta técnica de encapsular una excepción cuando un miembro de un parámetro produce una excepción que no se pudo prever antes de llamar al miembro.

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. 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 un establecedor de propiedades, [ParamName](#) debe establecerse en `value`.

Los métodos públicos y protegidos generan excepciones siempre que no puedan finalizar sus funciones previstas. La clase de excepciones generada es la excepción más específica disponible que se ajuste a las condiciones de 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:

- No use excepciones para cambiar el flujo de un programa como parte de la ejecución normal. Use excepciones para notificar y controlar 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](#) ni [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.

Excepciones en métodos que devuelven tareas

Los métodos declarados con el modificador `async` tienen algunas consideraciones especiales cuando se trata de excepciones. Las excepciones iniciadas en un método `async` se almacenan en la tarea devuelta y no surgen hasta que, por ejemplo, se espera la tarea. Para obtener más información sobre las excepciones almacenadas, consulte [Excepciones asincrónicas](#).

Se recomienda validar los argumentos e iniciar las excepciones correspondientes, como [ArgumentException](#) y [ArgumentNullException](#), antes de escribir las partes asincrónicas de los métodos. Es decir, estas excepciones de validación deben surgir sincrónicamente antes de que se inicie el trabajo. El siguiente fragmento de código muestra un ejemplo en el que, si se inician las excepciones, las excepciones [ArgumentException](#) surgirían

sincrónicamente, mientras que el `InvalidOperationException` se almacenaría en la tarea devuelta.

C#

```
// Non-async, task-returning method.  
// Within this method (but outside of the local function),  
// any thrown exceptions emerge synchronously.  
public static Task<Toast> ToastBreadAsync(int slices, int toastTime)  
{  
    if (slices < 1 or > 4)  
    {  
        throw new ArgumentException(  
            "You must specify between 1 and 4 slices of bread.",  
            nameof(slices));  
    }  
  
    if (toastTime < 1)  
    {  
        throw new ArgumentException(  
            "Toast time is too short.", nameof(toastTime));  
    }  
  
    return ToastBreadAsyncCore(slices, toastTime);  
  
    // Local async function.  
    // Within this function, any thrown exceptions are stored in the task.  
    static async Task<Toast> ToastBreadAsyncCore(int slices, int time)  
    {  
        for (int slice = 0; slice < slices; slice++)  
        {  
            Console.WriteLine("Putting a slice of bread in the toaster");  
        }  
        // Start toasting.  
        await Task.Delay(time);  
  
        if (time > 2_000)  
        {  
            throw new InvalidOperationException("The toaster is on fire!");  
        }  
  
        Console.WriteLine("Toast is ready!");  
  
        return new Toast();  
    }  
}
```

Definición de 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 tres constructores: un constructor sin parámetros, uno que establezca la propiedad de mensaje y otro que establezca las propiedades [Message](#) y [InnerException](#). Por ejemplo:

C#

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
        base(message, inner) { }
}
```

Agregue propiedades nuevas a la clase de excepción cuando los datos que proporcionan sean 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

- [Jerarquía de excepciones](#)



Colaborar con nosotros en
[GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

nuestra guía para
colaboradores.

☒ Proporcionar comentarios sobre
el producto

Excepciones generadas por el compilador

Artículo • 05/06/2023

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
ArithmetiException	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 tiempo de ejecución, en una conversión explícita de un tipo base a una interfaz o a un tipo derivado.
NullReferenceException	Se inicia cuando se intenta hacer referencia a un objeto cuyo valor es <code>null</code> .
OutOfMemoryException	Se inicia cuando se produce un error al intentar asignar memoria con el operador <code>new</code> . Esta excepción indica que se ha agotado la memoria disponible para el entorno compatible con Common Language Runtime.
OverflowException	Se inicia cuando se desborda una operación aritmética en un contexto <code>checked</code> .
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 <code>catch</code> compatible para capturarla.

Vea también

- Instrucciones para el control de excepciones

Convenciones y reglas de nomenclatura de identificadores de C#

Artículo • 27/11/2023

Un **identificador** es el nombre que se asigna a un tipo (clase, interfaz, struct, delegado o enumeración), miembro, variable o espacio de nombres.

Reglas de nomenclatura

Los identificadores válidos deben seguir estas reglas. El compilador de C# genera un error para cualquier identificador que no siga estas reglas:

- Los identificadores deben comenzar con una letra o un carácter de subrayado (_).
- 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 [artículo sobre identificadores en la Especificación del lenguaje C#](#).

Convenciones de nomenclatura

Además de las reglas, las convenciones para los nombres de identificador se usan en todas las API de .NET. Estas convenciones proporcionan coherencia para los nombres, pero el compilador no las aplica. Puede usar otras convenciones en los proyectos.

Por convención, los programas de C# usan `PascalCase` para nombres de tipo, espacios de nombres y todos los miembros públicos. Además, el equipo `dotnet/docs` usa las siguientes convenciones, adoptadas del [estilo de codificación del equipo en tiempo de ejecución de .NET](#):

- 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 deben contener dos caracteres de subrayado (`__`) consecutivos. Esos nombres están reservados para los identificadores generados por el compilador.
- Use nombres con significado y descriptivos para las variables, los métodos y las clases.
- Prefiere la claridad sobre la brevedad.
- Use PascalCase para los nombres de clase y los nombres de método.
- Use camelCase para los argumentos de método, las variables locales y los campos privados.
- Use PascalCase para los nombres de las constantes, tanto campos como constantes locales.
- Los campos de instancia privada comienzan con un carácter de subrayado (`_`).
- Los campos estáticos comienzan por `s_`. Esta convención no es el comportamiento predeterminado de Visual Studio ni forma parte de las instrucciones de diseño de Framework, pero se puede configurar en `editorconfig`.
- Evite usar abreviaturas o acrónimos en los nombres, excepto las abreviaturas ampliamente conocidas y aceptadas.
- Use espacios de nombres con significado y descriptivos que siguen la notación inversa del nombre de dominio.
- Elija nombres de ensamblado que representen el propósito principal del ensamblado.
- Evite usar nombres de una sola letra, excepto los contadores de bucle simples. Además, los ejemplos de sintaxis que describen la sintaxis de las construcciones de C# suelen usar los siguientes nombres de una sola letra que coinciden con la convención usada en la [especificación del lenguaje C#](#). Los ejemplos de sintaxis son una excepción a la regla.
 - Se usa `S` para las estructuras, `C` para las clases.
 - Se usa `M` para los métodos.
 - Se usa `V` para las variables, `P` para los parámetros.

- Use `r` para los parámetros `ref`.

💡 Sugerencia

Puede aplicar las convenciones de nomenclatura relativas a mayúsculas, prefijos, sufijos y separadores de palabras mediante **reglas de nomenclatura de estilo de código**.

En los ejemplos siguientes, las instrucciones relativas a los elementos marcados `public` son también aplicables cuando se trabaja con elementos `protected` y `protected internal`, los cuales están diseñados para ser visibles para los autores de llamadas externas.

Caso Pascal

Use la grafía Pascal ("PascalCasing") al asignar un nombre a los tipos `class`, `Interface`, `struct` o `delegate`.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{}
```

C#

```
public delegate void DelegateType(string message);
```

Al asignar un nombre a `interface`, use la grafía Pascal además de agregar el prefijo `I` al nombre. Este prefijo indica claramente a los consumidores que es un elemento `interface`.

C#

```
public interface IWorkerQueue
{
}
```

Al asignar nombres `public` a los miembros de los tipos, como campos, propiedades o eventos, use la grafía pascal. Además, use mayúsculas y minúsculas pascales para todos los métodos y funciones locales.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

Al escribir registros posicionales, use la grafía Pascal para los parámetros, ya que son las propiedades públicas del registro.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

Para más información sobre los registros posicionales, consulte [Sintaxis posicional para la definición de propiedades](#).

Grafía Camel

Use la grafía Camel ("camelCasing") al asignar nombres a los campos `private` o `internal`, y ponga los prefijos `_`. Use la grafía camel al asignar nombres a las variables locales, incluidas las instancias de un tipo delegado.

```
C#  
  
public class DataService  
{  
    private IWorkerQueue _workerQueue;  
}
```

💡 Sugerencia

Al editar código de C# que sigue estas convenciones de nomenclatura en un IDE que admite la finalización de instrucciones, al escribir `_` se mostrarán todos los miembros con ámbito de objeto.

Al trabajar con campos `static` que sean `private` o `internal`, use el prefijo `s_` y, para el subproceso estático, use `t_`.

```
C#  
  
public class DataService  
{  
    private static IWorkerQueue s_workerQueue;  
  
    [ThreadStatic]  
    private static TimeSpan t_timeSpan;  
}
```

Al escribir parámetros de método, use la grafía Camel.

```
C#  
  
public T SomeMethod<T>(int someNumber, bool isValid)  
{  
}
```

Para obtener más información sobre las convenciones de nomenclatura de C#, consulte el [estilo de codificación del equipo en tiempo de ejecución de .NET](#).

Instrucciones de nomenclatura de los parámetros de tipo

Las instrucciones siguientes se aplican a los parámetros de tipo en los parámetros de tipo genérico. Los parámetros de tipo son los marcadores de posición de los argumentos de un tipo genérico o un método genérico. Puede obtener más información sobre [los parámetros de tipo genéricos](#) en la guía de programación de C#.

- **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.

```
./snippets/coding-conventions
```

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- **Consideré** usar `T` como nombre del parámetro de tipo para los tipos con un parámetro de tipo de una sola letra.

```
./snippets/coding-conventions
```

```
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.

```
./snippets/coding-conventions
```

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- **Consideré** 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.

Convenciones de nomenclatura extra

- En ejemplos que no incluyan [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.

C#

```
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.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Convenciones de código de C# comunes

Artículo • 17/08/2023

Un estándar de código es esencial para mantener la legibilidad, la coherencia y la colaboración del código dentro de un equipo de desarrollo. Seguir las prácticas del sector y las directrices establecidas ayudan a garantizar que el código sea más fácil de comprender, mantener y ampliar. La mayoría de los proyectos aplican un estilo coherente a través de convenciones de código. Los proyectos [dotnet/docs](#) y [dotnet/samples](#) no son una excepción. En esta serie de artículos, aprenderá nuestras convenciones de codificación y las herramientas que usamos para aplicarlas. Puede tomar nuestras convenciones tal como están o modificarlas para satisfacer las necesidades de su equipo.

Elegimos nuestras convenciones en función de los siguientes objetivos:

1. *Corrección*: nuestros ejemplos se copian y pegan en las aplicaciones. Esperamos que, por lo tanto, debemos hacer que el código sea resistente y correcto, incluso después de varias modificaciones.
2. *Enseñanza*: el propósito de nuestras muestras es enseñar a todos los objetos .NET y C#. Por ese motivo, no se aplican restricciones a ninguna característica de lenguaje o API. En su lugar, esos ejemplos enseñan cuando una característica es una buena opción.
3. *Coherencia*: los lectores esperan una experiencia coherente en todo nuestro contenido. Todos los ejemplos deben ajustarse al mismo estilo.
4. *Adopción*: actualizamos de forma agresiva nuestros ejemplos para usar nuevas características de lenguaje. Esa práctica genera conciencia de las nuevas características y hace que sean más familiares para todos los desarrolladores de C#.

ⓘ Importante

Microsoft usa las instrucciones para desarrollar ejemplos y documentación. Se adoptaron a partir de las instrucciones de [estilo de codificación de .NET Runtime y C#](#) y guía de [compilador C# \(roslyn\)](#). Hemos elegido esas directrices porque se han probado durante varios años de desarrollo de código abierto. Han ayudado a los miembros de la comunidad a participar en los proyectos de tiempo de ejecución y compilador. Están diseñados para ser un ejemplo de convenciones comunes de C# y no una lista autoritativa (consulte [Directrices de diseño del marco](#) para ello).

Los objetivos de *enseñanza* y *adopción* son los motivos por los que la convención de codificación de documentos difiere de las convenciones del entorno de ejecución y del compilador. Tanto el tiempo de ejecución como el compilador tienen métricas de rendimiento estrictas para las rutas de acceso activas. Muchas otras aplicaciones no. Nuestro objetivo de *enseñanza* exige que no prohibimos ninguna construcción. En su lugar, los ejemplos muestran cuándo se deben usar construcciones. Actualizamos los ejemplos de forma más agresiva que la mayoría de las aplicaciones de producción. Nuestro objetivo de *adopción* exige que se muestre código que debe escribir hoy, incluso cuando el código escrito el año pasado no necesita cambios.

En este artículo se explica nuestra guía. Las directrices han evolucionado con el tiempo y encontrará ejemplos que no siguen nuestras directrices. Agradecemos las solicitudes de incorporación de cambios que incluyan esos ejemplos en cumplimiento o incidencias que centran nuestra atención en las muestras que debemos actualizar. Nuestras directrices son de código abierto y agradecemos las solicitudes de incorporación de cambios y de incidencias. Sin embargo, si su envío cambiaría estas recomendaciones, abra primero una incidencia para discutirlo. Le animamos a usar nuestras directrices o adaptarlas a sus necesidades.

Herramientas y analizadores

Las herramientas pueden ayudar a su equipo a aplicar sus estándares. Puede habilitar cualquiera de las [herramientas de análisis de código](#) para aplicar las reglas que prefiera. También puede crear una [editorconfig](#) para que Visual Studio aplique automáticamente las directrices de estilo. Puede empezar usando [el dotnet/docs ↗](#) para usar nuestro estilo como punto de partida.

Estas herramientas facilitan a su equipo adoptar sus directrices preferidas. Visual Studio aplica las reglas de todos los archivos de `.editorconfig` en el ámbito para dar formato al código. Puede usar varios conjuntos de reglas para aplicar estándares corporativos, estándares de equipo e incluso estándares de proyecto pormenorizados.

Las herramientas de análisis de código configuradas generan advertencias y diagnósticos cuando se infringen sus reglas. Configure las reglas que desea aplicar al proyecto. A continuación, cada compilación de CI notifica a los desarrolladores cuando infringen cualquiera de las reglas.

Convenciones de lenguaje

En las secciones siguientes se describen las prácticas que sigue el equipo de documentación de .NET para preparar las muestras y ejemplos de código. En general, siga estos procedimientos:

- Use las características modernas del lenguaje y las versiones de C# siempre que sea posible.
- Evite construcciones de lenguaje obsoletas.
- Solo trabaje con las excepciones que se pueden controlar correctamente; evite trabajar sobre excepciones genéricas.
- Use tipos de excepción específicos para proporcionar mensajes de error significativos.
- Use consultas y métodos LINQ para la manipulación de recopilación para mejorar la legibilidad del código.
- Use una programación asíncrona con `async` y `await` para operaciones enlazadas a E/S.
- Tenga cuidado con los interbloqueos y use `Task.ConfigureAwait` cuando corresponda.
- Use las palabras clave del lenguaje para los tipos de datos en lugar de los tipos de tiempo de ejecución. Por ejemplo, use `string` en vez de `System.String` o `int` en lugar de `System.Int32`.
- 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`. Las excepciones son para la documentación específica de los tipos de datos sin firmar.
- Use `var` solo cuando un lector pueda deducir el tipo de la expresión. Los lectores ven nuestros ejemplos en la plataforma de documentos. No tienen sugerencias pasando el ratón por encima ni las herramientas que muestran el tipo de variables.
- Escriba el código pensando en su claridad y simplicidad.
- Evite la lógica de código demasiado compleja y enrevesada.

Se siguen instrucciones más específicas.

Datos de cadena

- Use [interpolación de cadenas](#) para concatenar cadenas cortas, como se muestra en el código siguiente.

C#

```
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 `System.Text.StringBuilder`.

Matrices

- Utilice sintaxis concisa para inicializar las matrices en la línea de declaración. En el siguiente ejemplo, no puede utilizar `var` en lugar de `string[]`.

C#

- Si usa la creación de instancias explícita, puede usar `var`.

C#

Delegados

- Use `Func<>` y `Action<>` en lugar de definir tipos de delegado. En una clase, defina el método delegado.

```
C#  
  
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");  
  
Action<string, string> actionExample2 = (x, y) =>  
    Console.WriteLine($"x is: {x}, y is {y}");  
  
Func<string, int> funcExample1 = x => Convert.ToInt32(x);  
  
Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Llame al método con la firma definida por el delegado `Func<>` o `Action<>`.

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- Si crea instancias de un tipo de delegado, utilice la sintaxis concisa. En una clase, defina el tipo de delegado y un método que tenga una firma coincidente.

C#

```
public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- Cree una instancia del tipo de delegado y llámela. La siguiente declaración muestra la sintaxis condensada.

C#

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- La siguiente declaración utiliza la sintaxis completa.

C#

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

Instrucciones `try-catch` y `using` en el control de excepciones

- Use una instrucción `try-catch` en la mayoría de los casos de control de excepciones.

C#

```

static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmetiException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        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](#).

En el ejemplo siguiente, la instrucción [try-finally](#) solo llama a [Dispose](#) en el bloque [finally](#).

```

C#

Font bodyStyle = new Font("Arial", 10.0f);
try
{
    byte charset = bodyStyle.GdiCharSet;
}
finally
{
    if (bodyStyle != null)
    {
        ((IDisposable)bodyStyle).Dispose();
    }
}

```

Puede hacer lo mismo con una instrucción [using](#).

```

C#

using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}

```

Use la nueva [sintaxis using](#) que no requiere corchetes:

C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

Operadores `&&` y `||`

- Use `&&` en vez de `&` y `||` en vez de `|` cuando realice comparaciones, como se muestra en el ejemplo siguiente.

C#

```
Console.WriteLine("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Si el divisor es 0, la segunda cláusula de la instrucción `if` produciría un error en tiempo de ejecución. Pero el operador `&&` se cortocircuita cuando la primera expresión es falsa. Es decir, no evalúa la segunda expresión. El operador `&` evaluaría ambas, lo que provocaría un error en tiempo de ejecución cuando `divisor` es 0.

Operador `new`

- Use una de las formas concisas de creación de instancias de objeto, tal como se muestra en las declaraciones siguientes. En el segundo ejemplo se muestra la sintaxis que está disponible a partir de C# 9.

C#

```
var firstExample = new ExampleClass();
```

C#

```
ExampleClass instance2 = new();
```

Las declaraciones anteriores son equivalentes a la siguiente declaración.

C#

```
ExampleClass secondExample = new ExampleClass();
```

- Use inicializadores de objeto para simplificar la creación de objetos, tal y como se muestra en el ejemplo siguiente.

C#

```
var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

En el ejemplo siguiente se establecen las mismas propiedades que en el ejemplo anterior, pero no se utilizan inicializadores.

C#

```
var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

Control de eventos

- Use una expresión lambda para definir un controlador de eventos que no es necesario quitar más tarde:

C#

```
public Form2()
{
    this.Click += (s, e) =>
    {
        MessageBox.Show(
            ((MouseEventArgs)e).Location.ToString());
    };
}
```

La expresión lambda acorta la siguiente definición tradicional.

C#

```
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.

C#

```
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.

C#

```
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 ambiguas 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.

C#

```
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. Esta guía sobre la escritura implícita en consultas LINQ invalida las reglas generales de [las variables locales con tipo implícito](#). Las consultas LINQ suelen usar proyecciones que crean tipos anónimos. Otras expresiones de consulta crean resultados con tipos genéricos anidados. Las variables con tipo implícito suelen ser más legibles.

C#

```
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.

C#

```
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.

C#

```
var scoreQuery = from student in students
                  from score in student.Scores!
                  where score > 90
                  select new { Last = student.LastName, score };
```

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 tarea.

C#

```
var message = "This is clearly a string.";
var currentTemperature = 27;
```

- No use **var** cuando el tipo no sea evidente desde el lado derecho de la tarea. No asuma que el tipo está claro a partir de un nombre de método. Se considera que un tipo de variable es claro si es un operador **new**, una conversión explícita o tarea para un valor literal.

C#

```
int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- No use nombres de variable para especificar el tipo de la variable. Puede no ser correcto. En su lugar, use el tipo para especificar el tipo y use el nombre de la variable para indicar la información semántica de la variable. En el ejemplo siguiente se debe usar **string** para el tipo y algo parecido a **iterations** para indicar el significado de la información leída desde la consola.

C#

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Evite el uso de `var` en lugar de `dynamic`. Use `dynamic` cuando desee la inferencia de tipos en tiempo de ejecución. Para obtener más información, vea [Uso de tipo dinámico \(Guía de programación de C#\)](#).
 - Use la escritura implícita de la variable de bucle en bucles `for`.

En el ejemplo siguiente se usan tipos implícitos en una instrucción `for`.

- No use tipos implícitos para determinar el tipo de la variable de bucle en bucles `foreach`. En la mayoría de los casos, el tipo de elementos de la colección no es inmediatamente obvio. El nombre de la colección no debe servir únicamente para inferir el tipo de sus elementos.

En el ejemplo siguiente se usan tipos explícitos en una instrucción `foreach`.

```
C#  
  
foreach (char ch in laugh)  
{  
    if (ch == 'h')  
        Console.WriteLine("H");  
    else  
        Console.Write(ch);  
}  
Console.WriteLine();
```

- use el tipo implícito para las secuencias de resultados en las consultas LINQ. En la sección sobre [LINQ](#) se explica que muchas consultas LINQ dan lugar a tipos anónimos en los que se deben usar tipos implícitos. Otras consultas dan como resultado tipos genéricos anidados en los que `var` es más legible.

! Nota

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.

Algunos de nuestros ejemplos explican *el tipo natural* de una expresión. Esos ejemplos deben usar `var` para que el compilador elija el tipo natural. Aunque esos ejemplos son menos obvios, se requiere el uso de `var` para el ejemplo. El texto debe explicar el comportamiento.

Colocación de directivas `using` fuera de la declaración del espacio de nombres

Cuando una directiva `using` está fuera de la declaración de un espacio de nombres, ese espacio de nombres importado es su nombre completo. El nombre completo es más claro. Cuando la directiva `using` está dentro del espacio de nombres, puede ser relativa al espacio de nombres o su nombre completo.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Se supone que hay una referencia (directa o indirecta) a la clase `WaitUntil`.

Ahora vamos a cambiarla ligeramente:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
```

```
        WaitUntil wait = WaitUntil.Completed;
        // ...
    }
}
```

Y se compila hoy. Y mañana. Sin embargo, en algún momento de la próxima semana, el código anterior (sin modificar) produce dos errores:

Consola

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

Una de las dependencias ha introducido esta clase en un espacio de nombres y, después, finaliza con `.Azure`:

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
            return null; }
    }
}
```

Una directiva `using` colocada dentro de un espacio de nombres es contextual y complica la resolución de nombres. En este ejemplo, es el primer espacio de nombres que encuentra.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Si se agrega un nuevo espacio de nombres que coincida con `CoolStuff.Azure` o `CoolStuff.AwesomeFeature.Azure`, se tomaría como coincidencia antes que el espacio de nombres global `Azure`. Para resolverlo, agregue el modificador `global::` a la declaración `using`. Sin embargo, es más fácil colocar declaraciones `using` fuera del espacio de nombres.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Guía de estilo

En general, use el siguiente formato para ejemplos de código:

- Use cuatro espacios para la sangría. No use pestañas.
- Alinee el código de forma coherente para mejorar la legibilidad.
- Limite las líneas a 65 caracteres para mejorar la legibilidad del código en documentos, especialmente en pantallas móviles.
- Divida instrucciones largas en varias líneas para mejorar la claridad.
- Use el estilo "Allman" para llaves: llaves de apertura y cierre en su propia línea nueva. Las llaves se alinean con el nivel de sangría actual.
- Los saltos de línea deben producirse antes de los operadores binarios, si es necesario.

Estilo de comentario

- Use comentarios de una sola línea (`//`) para las explicaciones breves.
- Evite los comentarios de varias líneas (`/* */`) para las explicaciones más largas. Los comentarios no se localizan. En su lugar, las explicaciones más largas se encuentran en el artículo complementario.
- 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.

```
C#
```

```
// The following declaration creates a query. It does not run  
// the query.
```

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.

```
C#
```

```
if ((startX > endX) && (startX > previousX))  
{  
    // Take appropriate action.  
}
```

Las excepciones son cuando el ejemplo explica la precedencia de operador o expresión.

Seguridad

Siga las instrucciones de [Instrucciones de codificación segura](#).

Procedimiento para mostrar argumentos de la línea de comandos

Artículo • 08/06/2023

Los argumentos proporcionados para un archivo ejecutable en la línea de comandos son accesibles en [instrucciones de nivel superior](#) o mediante un parámetro opcional para `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"

ⓘ Nota

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 la línea de comandos pasados a una aplicación de la línea de comandos. La salida que se muestra corresponde a la primera entrada de la tabla anterior.

C#

```
// 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]
*/
```

Consulte también

- [Introducción a System.CommandLine](#)
- [Tutorial: Introducción a System.CommandLine](#)

Exploración de la programación orientada a objetos con clases y objetos

Artículo • 05/06/2023

En este tutorial, creará una aplicación de consola y conocerá las características básicas orientadas a objetos que forman parte del lenguaje C#.

Requisitos previos

- Se recomienda tener [Visual Studio](#) para Windows o Mac. Puede descargar una versión gratuita desde la [página de descargas de Visual Studio](#). Visual Studio incluye el SDK de .NET.
- También se puede usar el editor de [Visual Studio Code](#). Deberá instalar el [SDK de .NET](#) más reciente por separado.
- Si prefiere usar otro editor, deberá instalar el [SDK de .NET](#) más reciente.

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:

C#

```
// See https://aka.ms/new-console-template for more information
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 *BankAccount.cs* en el directorio *Classes*.

Este archivo va a contener la definición de una *cuenta bancaria*. La programación orientada a objetos organiza el código creando 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 generar 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**. Asígnale el nombre *BankAccount.cs*. Agregue el código siguiente al archivo *BankAccount.cs*:

C#

```
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 quiere crear. Todo lo que se encuentra entre `{` y `}` después de la declaración de clase define el estado y el comportamiento de la clase. La clase `BankAccount` cuenta con cinco **miembros**. 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.

Para crear un objeto de tipo `BankAccount`, es necesario definir un **constructor** que asigne esos valores. Un **constructor** es un miembro que tiene el mismo nombre que la clase. Se usa 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`.

C#

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

En el código anterior se identifican las propiedades del objeto que se está construyendo mediante la inclusión del calificador `this`. Ese calificador suele ser opcional y se omite. También podría haber escrito lo siguiente:

C#

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

El calificador `this` solo es necesario cuando una variable o un parámetro local tiene el mismo nombre que el campo o la propiedad. El calificador `this` se omite en el resto de este artículo, a menos que sea necesario.

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):

C#

```
using Classes;

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 **Depurar**. 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 empezar es 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`:

C#

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

`accountNumberSeed` es 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 lo comparten 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`:

C#

```
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. Hacer un seguimiento de cada transacción ofrece algunas ventajas con respecto a limitarse a 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, nos aseguramos de que todos los errores de una única transacción que se solucionen se reflejarán correctamente en el saldo cuando se haga el siguiente cálculo.

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

```
C#  
  
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)  
    {  
        Amount = amount;  
        Date = date;  
        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*:

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

Ahora vamos a calcular correctamente `Balance`. El saldo actual se puede averiguar si se suman los valores de todas las transacciones. Como el código es actual, solo puede obtener el saldo inicial de la cuenta, así que tiene que actualizar la propiedad `Balance`.

Reemplace la línea `public decimal Balance { get; }` de *BankAccount.cs* con el código siguiente:

```
C#  
  
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 aplicarán las dos reglas finales: el saldo inicial debe ser positivo, y ningún reintegro debe generar un saldo negativo.

Las reglas presentan 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 no es mayor que 0. El método `MakeWithdrawal` genera una excepción si el importe del reintegro no es mayor que 0 o si el procesamiento de la operación tiene como resultado un saldo negativo: Agregue el código siguiente después de la declaración de la lista `allTransactions`:

```
C#  
  
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` produce 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:

C#

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

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

`DateTime.Now` es una propiedad que devuelve la fecha y hora actuales. Para probar este código, agregue algunos depósitos y reintegros en el método `Main`, siguiendo el código con el que se crea un elemento `BankAccount`:

C#

```
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 intentando crear una cuenta con un saldo negativo. Agregue el código siguiente después del código anterior que acaba de agregar:

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

Use la [instrucción try-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 antes de la declaración de `invalidAccount` en el método `Main`:

```
C#  
  
// 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`:

```
C#
```

```
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();
}
```

En el historial se 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`:

C#

```
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](#).

Puede continuar con el tutorial de la [programación orientada a objetos](#).

Puede aprender más sobre estos conceptos en los artículos siguientes:

- [Instrucciones de selección](#)
- [Instrucciones de iteración](#)

Programación orientada a objetos (C#)

Artículo • 08/06/2023

C# es un lenguaje de programación orientado a objetos. Los cuatro principios básicos de la programación orientada a objetos son:

- *Abstracción*: modelar los atributos e interacciones pertinentes de las entidades como clases para definir una representación abstracta de un sistema.
- *Encapsulación*: ocultar el estado interno y la funcionalidad de un objeto y permitir solo el acceso a través de un conjunto público de funciones.
- *Herencia*: capacidad de crear nuevas abstracciones basadas en abstracciones existentes.
- *Polimorfismo*: capacidad de implementar propiedades o métodos heredados de maneras diferentes en varias abstracciones.

En el tutorial anterior, [Introducción a las clases](#) se trató la *abstracción* y la *encapsulación*. La clase `BankAccount` proporcionó una abstracción para el concepto de una cuenta bancaria. Puede modificar su implementación sin que afecte para nada al código que usó la clase `BankAccount`. Las clases `BankAccount` y `Transaction` proporcionan encapsulación de los componentes necesarios para describir esos conceptos en el código.

En este tutorial, ampliará la aplicación para hacer uso de la *herencia* y el *polimorfismo* para agregar nuevas características. También agregará características a la clase `BankAccount`, aprovechando las técnicas de *abstracción* y *encapsulación* que aprendió en el tutorial anterior.

Creación de diferentes tipos de cuentas

Después de compilar este programa, recibirá solicitudes para agregarle características. Funciona bien en situaciones en las que solo hay un tipo de cuenta bancaria. Con el tiempo, las necesidades cambian y se solicitan tipos de cuenta relacionados:

- Una cuenta que devenga intereses que genera beneficios al final de cada mes.
- Una línea de crédito que puede tener un saldo negativo, pero cuando sea así, se producirá un cargo por intereses cada mes.
- Una cuenta de tarjeta de regalo de prepago que comienza con un único depósito y solo se puede liquidar. Se puede recargar una vez al principio de cada mes.

Todas estas cuentas diferentes son similares a la clase `BankAccount` definida en el tutorial anterior. Podría copiar ese código, cambiar el nombre de las clases y realizar

modificaciones. Esa técnica funcionaría a corto plazo, pero a la larga supondría más trabajo. Cualquier cambio se copiará en todas las clases afectadas.

En su lugar, puede crear nuevos tipos de cuenta bancaria que hereden métodos y datos de la clase `BankAccount` creada en el tutorial anterior. Estas clases nuevas pueden extender la clase `BankAccount` con el comportamiento específico necesario para cada tipo:

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Cada una de estas clases *hereda* el comportamiento compartido de su *clase base* compartida, la clase `BankAccount`. Escriba las implementaciones para la funcionalidad nueva y diferente en cada una de las *clases derivadas*. Estas clases derivadas ya tienen todo el comportamiento definido en la clase `BankAccount`.

Es recomendable crear cada clase nueva en un archivo de código fuente diferente. En [Visual Studio](#), puede hacer clic con el botón derecho en el proyecto y seleccionar *Agregar clase* para agregar una clase nueva en un archivo nuevo. En [Visual Studio Code](#), seleccione *Archivo* y luego *Nuevo* para crear un nuevo archivo de código fuente. En cualquier herramienta, ponga un nombre al archivo que coincida con la clase: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs* y *GiftCardAccount.cs*.

Cuando cree las clases como se muestra en el ejemplo anterior, observará que ninguna de las clases derivadas se compila. La inicialización de un objeto es responsabilidad de un constructor. Un constructor de clase derivada debe inicializar la clase derivada y proporcionar instrucciones sobre cómo inicializar el objeto de la clase base incluido en la clase derivada. Normalmente, se produce una inicialización correcta sin ningún código adicional. La clase `BankAccount` declara un constructor público con la siguiente firma:

C#

```
public BankAccount(string name, decimal initialBalance)
```

El compilador no genera un constructor predeterminado al definir un constructor. Esto significa que cada clase derivada debe llamar explícitamente a este constructor. Se declara un constructor que puede pasar argumentos al constructor de la clase base. En el código siguiente se muestra el constructor de `InterestEarningAccount`:

C#

```
public InterestEarningAccount(string name, decimal initialBalance) :  
    base(name, initialBalance)  
{  
}
```

Los parámetros de este nuevo constructor coinciden con el tipo de parámetro y los nombres del constructor de clase base. Utilice la sintaxis de `: base()` para indicar una llamada a un constructor de clase base. Algunas clases definen varios constructores, y esta sintaxis le permite elegir el constructor de clase base al que llama. Una vez que haya actualizado los constructores, puede desarrollar el código para cada una de las clases derivadas. Los requisitos para las clases nuevas se pueden indicar de la siguiente manera:

- Una cuenta que devenga intereses:
 - obtendrá un crédito del 2 % del saldo a final de mes.
- Una línea de crédito:
 - puede tener un saldo negativo, pero no mayor en valor absoluto que el límite de crédito.
 - Generará un cargo por intereses cada mes en el que el saldo final del mes no sea 0.
 - Generará un cargo por cada retirada que supere el límite de crédito.
- Una cuenta de tarjeta regalo:
 - Se puede recargar con una cantidad especificada una vez al mes, el último día del mes.

Puede ver que los tres tipos de cuenta tienen una acción que tiene lugar al final de cada mes. Sin embargo, cada tipo de cuenta realiza diferentes tareas. Utiliza el *polimorfismo* para implementar este código. Cree un método `virtual` único en la clase `BankAccount`:

C#

```
public virtual void PerformMonthEndTransactions() { }
```

El código anterior muestra cómo se usa la palabra clave `virtual` para declarar un método en la clase base para el que una clase derivada puede proporcionar una

implementación diferente. Un método `virtual` es un método en el que cualquier clase derivada puede optar por volver a implementar. Las clases derivadas usan la palabra clave `override` para definir la nueva implementación. Normalmente, se hace referencia a este proceso como "reemplazar la implementación de la clase base". La palabra clave `virtual` especifica que las clases derivadas pueden invalidar el comportamiento.

También puede declarar métodos `abstract` en los que las clases derivadas deben reemplazar el comportamiento. La clase base no proporciona una implementación para un método `abstract`. A continuación, debe definir la implementación de dos de las nuevas clases que ha creado. Empiece por `InterestEarningAccount`:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Agregue el código siguiente a `LineOfCreditAccount`. El código niega el saldo para calcular un cargo de interés positivo que se retira de la cuenta:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

La clase `GiftCardAccount` necesita dos cambios para implementar su funcionalidad de fin de mes. En primer lugar, modifique el constructor para incluir una cantidad opcional para agregar cada mes:

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal
```

```
monthlyDeposit = 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

El constructor proporciona un valor predeterminado para el valor `monthlyDeposit`, por lo que los llamadores pueden omitir `0` para ningún ingreso mensual. A continuación, invalide el método `PerformMonthEndTransactions` para agregar el depósito mensual, si se estableció en un valor distinto de cero en el constructor:

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

La invalidación aplica el conjunto de depósitos mensual en el constructor. Agregue el código siguiente al método `Main` para probar estos cambios en `GiftCardAccount` y en `InterestEarningAccount`:

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending
money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Verifique los resultados. Ahora, agregue un conjunto similar de código de prueba para `LineOfCreditAccount`:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
```

```
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly  
advance");  
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");  
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for  
repairs");  
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on  
repairs");  
lineOfCredit.PerformMonthEndTransactions();  
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Al agregar el código anterior y ejecutar el programa, verá algo parecido al siguiente error:

Consola

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit  
must be positive (Parameter 'amount')  
at OOPProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date,  
String note) in BankAccount.cs:line 42  
at OOPProgramming.BankAccount..ctor(String name, Decimal initialBalance)  
in BankAccount.cs:line 31  
at OOPProgramming.LineOfCreditAccount..ctor(String name, Decimal  
initialBalance) in LineOfCreditAccount.cs:line 9  
at OOPProgramming.Program.Main(String[] args) in Program.cs:line 29
```

ⓘ Nota

La salida real incluye la ruta de acceso completa a la carpeta con el proyecto. Los nombres de carpeta se omitieron para ser más breves. Además, dependiendo del formato del código, los números de línea pueden ser ligeramente diferentes.

Este código produce un error porque `BankAccount` supone que el saldo inicial debe ser mayor que 0. Otra suposición incorporada en la clase `BankAccount` es que el saldo no puede entrar en cifras negativas. Lo que sucede que es se rechazan las retiradas que provocan un descubierto en la cuenta. Ambas suposiciones deben cambiar. La línea de la cuenta de crédito comienza en 0, y generalmente tendrá un saldo negativo. Además, si un cliente retira demasiado dinero, generará un cargo. La transacción se acepta, solo que cuesta más. La primera regla se puede implementar agregando un argumento opcional al constructor `BankAccount` que especifica el saldo mínimo. El valor predeterminado es `0`. La segunda regla requiere un mecanismo que permita que las clases derivadas modifiquen el algoritmo predeterminado. En cierto sentido, la clase base "pregunta" al tipo derivado qué debe ocurrir cuando hay un descubierto. El comportamiento predeterminado es rechazar la transacción generando una excepción.

Comencemos agregando un segundo constructor que incluya un parámetro `minimumBalance` opcional. Este nuevo constructor se ocupa de todas las acciones que realiza el constructor existente. Además, establece la propiedad del saldo mínimo. Puede copiar el cuerpo del constructor existente, pero eso significa que dos ubicaciones cambiarán en el futuro. Lo que puede hacer es usar un *encadenamiento de constructores* para que un constructor llame a otro. En el código siguiente se muestran los dos constructores y el nuevo campo adicional:

C#

```
private readonly decimal _minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal
minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

En el código anterior se muestran dos técnicas nuevas. En primer lugar, el campo `minimumBalance` está marcado como `readonly`. Esto significa que el valor no se puede cambiar después de que se construya el objeto. Una vez que se crea `BankAccount`, `minimumBalance` no puede cambiar. En segundo lugar, el constructor que toma dos parámetros utiliza `: this(name, initialBalance, 0) { }` como su implementación. La expresión `: this()` llama al otro constructor, el que tiene tres parámetros. Esta técnica permite tener una única implementación para inicializar un objeto, aunque el código de cliente puede elegir uno de muchos constructores.

Esta implementación solo llama a `MakeDeposit` si el saldo inicial es mayor que `0`. Esto conserva la regla de que los depósitos deben ser positivos, pero permite que la cuenta de crédito se abra con un saldo de `0`.

Ahora que la clase `BankAccount` tiene un campo de solo lectura para el saldo mínimo, el último cambio es modificar la codificación rígida `0` a `minimumBalance` en el método `MakeWithdrawal`:

C#

```
if (Balance - amount < minimumBalance)
```

Después de extender la clase `BankAccount`, puede modificar el constructor `LineOfCreditAccount` para llamar al nuevo constructor base, como se muestra en el código siguiente:

C#

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

Observe que el constructor `LineOfCreditAccount` cambia el signo del parámetro `creditLimit` para que coincida con el significado del parámetro `minimumBalance`.

Diferentes reglas de descubierto

La última característica que se va a agregar permite a `LineOfCreditAccount` cobrar una cuota por sobrepasar el límite de crédito en lugar de rechazar la transacción.

Una técnica consiste en definir una función virtual en la que se implemente el comportamiento requerido. La clase `BankAccount` refactoriza el método `MakeWithdrawal` en dos métodos. El nuevo método realiza la acción especificada cuando la retirada toma el saldo por debajo del mínimo. El método `MakeWithdrawal` existente tiene el siguiente código:

C#

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

Reemplácelo por el código siguiente:

```
C#  
  
public void MakeWithdrawal(decimal amount, DateTime date, string note)  
{  
    if (amount <= 0)  
    {  
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of  
withdrawal must be positive");  
    }  
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance -  
amount < _minimumBalance);  
    Transaction? withdrawal = new(-amount, date, note);  
    _allTransactions.Add(withdrawal);  
    if (overdraftTransaction != null)  
        _allTransactions.Add(overdraftTransaction);  
}  
  
protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)  
{  
    if (isOverdrawn)  
    {  
        throw new InvalidOperationException("Not sufficient funds for this  
withdrawal");  
    }  
    else  
    {  
        return default;  
    }  
}
```

El método agregado es `protected`, lo que significa que solo se puede llamar desde clases derivadas. Esta declaración impide que otros clientes llamen al método. También es `virtual` para que las clases derivadas puedan cambiar el comportamiento. El tipo de valor devuelto es `Transaction?`. La anotación `?` indica que el método puede devolver `null`. Agregue la siguiente implementación en `LineOfCreditAccount` para cobrar una cuota cuando se supere el límite de retirada:

```
C#  
  
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>  
    isOverdrawn  
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")  
    : default;
```

El reemplazo devuelve una transacción de cuota cuando en la cuenta se produce un descubierto. Si la retirada no supera el límite, el método devuelve una transacción `null`.

Esto indica que no hay ninguna cuota. Pruebe estos cambios agregando el código siguiente al método `Main` en la clase `Program`:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Ejecute el programa y compruebe los resultados.

Resumen

Si se ha quedado bloqueado, puede consultar el origen de este tutorial [en el repositorio de GitHub](#).

En este tutorial se han mostrado muchas de las técnicas que se usan en la programación orientada a objetos:

- Usó la *abstracción* cuando definió clases para cada uno de los distintos tipos de cuenta. Esas clases describían el comportamiento de ese tipo de cuenta.
- Usó la *encapsulación* cuando mantuvo muchos detalles `private` en cada clase.
- Usó la *herencia* cuando aprovechó la implementación ya creada en la clase `BankAccount` para guardar el código.
- Usó el *polimorfismo* cuando creó métodos `virtual` que las clases derivadas podrían reemplazar para crear un comportamiento específico para ese tipo de cuenta.

Herencia en C# y .NET

Artículo • 05/06/2023

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

- Se recomienda tener [Visual Studio](#) para Windows o Mac. Puede descargar una versión gratuita desde la [página de descargas de Visual Studio](#). Visual Studio incluye el SDK de .NET.
- También se puede usar el editor de [Visual Studio Code](#). Deberá instalar el [SDK de .NET](#) más reciente por separado.
- Si prefiere usar otro editor, deberá instalar el [SDK de .NET](#) más reciente.

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 `--noRestore`.

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 visibles 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 privado `A._value` es visible en `A.B`. Pero 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".

```

public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}

// The example displays the following output:
//      10

```

- Los miembros **protégidos** 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**.

C#

```

public class A
{
    public void Method1()
    {

```

```

        // Method implementation.
    }

}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new ();
        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: "<miembro> no puede invalidar el miembro heredado <miembro> porque no está marcado como virtual, abstract ni override".

C#

```

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`.

C#

```
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 en el ejemplo siguiente se produce el error del compilador CS0527: "El tipo 'ValueType' en la lista de interfaz no es una interfaz". El mensaje de error indica que, aunque se pueden definir las interfaces que implementa una estructura, no se admite la herencia.

C#

```
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:

C#

```
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.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    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 (MemberInfo 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";
            }
            string 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`.

C#

```
public class EmptyClass
{ }

public class ClassNameExample
```

```

{
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//      EmptyClass

```

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.

ⓘ Nota

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`.

```
C#  
  
public class Automobile  
{  
    public Automobile(string make, string model, int year)  
    {  
        if (make == null)  
            throw new ArgumentNullException(nameof(make), "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(nameof(model), "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.

C#

```
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 clase o interfaz 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 sellada, se genera el error de compilador CS0509: "No puede derivar del tipo sellado <nombreDeTipo>".

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:

C#

```
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(nameof(value), "The
number of pages cannot be zero or negative.");
        _totalPages = value;
    }
}

public string GetPublicationDate()
{
    if (!_published)
        return "NYP";
    else
        return _datePublished.ToString("d");
}

public void Publish(DateTime datePublished)
{
    _published = true;
    _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:

C#

```

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`.

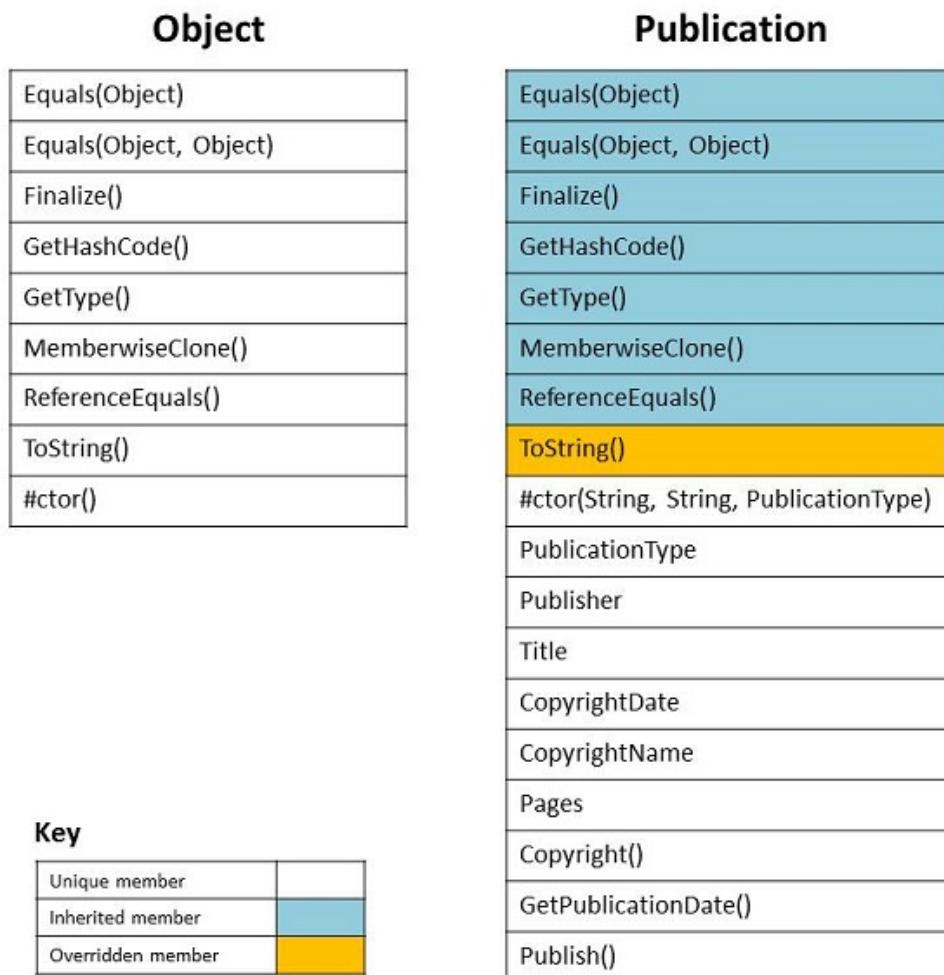
- Miembros relacionados con copyright

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`.

C#

```
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.");
            if (!ulong.TryParse(isbn, out _))
                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(nameof(price), "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)
    {
        if (obj is not Book book)
            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 pasarlalos como argumento a un método que espera un parámetro de tipo `Publication` o de tipo `Book`, como se muestra en el ejemplo siguiente.

C#

```
public class ClassExample
{
    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");
Console.WriteLine($"{book.Title} and {book2.Title} are the same
publication: " +
"${((Publication)book).Equals(book2)}");
}

public static void ShowPublicationInfo(Publication pub)
{
    string pubDate = pub.GetPublicationDate();
    Console.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 comparten 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 cuadrado. 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.

C#

```
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衍生一些 classes de `Shape` que representan formas concretas. El ejemplo siguiente define tres clases `Square`, `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.

C#

```
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`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (Shape shape in shapes)
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, " +
Diagonal: {rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                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
}
```

Procedimiento para convertir de forma segura mediante la coincidencia de patrones y los operadores is y as

Artículo • 07/04/2023

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.

C#

```
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);

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");
}
}

// Output:
// I am an animal.
// SuperNova is not a Mammal

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 { }

```

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:

C#

```

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(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(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:

C#

```
// Use the is operator to verify the type.  
// before performing a cast.  
Giraffe g = new();  
UseIsOperator(g);  
  
// Use the as operator and test for null  
// before referencing the variable.  
UseAsOperator(g);  
  
// Use pattern matching to test for null  
// before referencing the variable  
UsePatternMatchingIs(g);  
  
// Use the as operator to test  
// an incompatible type.  
SuperNova sn = new();  
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 is not 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 is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}

class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

```

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.

Tutorial: Uso de la coincidencia de patrones para compilar algoritmos basados en tipos y basados en datos

Artículo • 10/05/2023

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 a:

- ✓ 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

- Se recomienda tener [Visual Studio](#) para Windows o Mac. Puede descargar una versión gratuita desde la [página de descargas de Visual Studio](#). Visual Studio incluye el SDK de .NET.
- También se puede usar el editor de [Visual Studio Code](#). Deberá instalar el [SDK de .NET](#) más reciente por separado.
- Si prefiere usar otro editor, deberá instalar el [SDK de .NET](#) más reciente.

En este tutorial se da por supuesto que conoce bien C# y .NET, incluidos Visual Studio o la CLI de .NET.

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:

```
C#  
  
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 usar 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 resolver lo siguiente:

- 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`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

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 lo mismo que una instrucción `switch`) que prueba el [patrón de declaración](#). 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 `null` [patrón de constante](#) detecta si se pasa `null` a este método. El patrón `null` puede ser el último porque los otros patrones solo coinciden con un objeto no nulo del tipo correcto.

Puede probar este código con el código siguiente en `Program.cs`:

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;
```

```

using toll_calculator;

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 un [patrón de propiedad](#) en la misma expresión switch. Un patrón de propiedad compara un valor de propiedad con un valor constante. 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:

```
C#  
  
vehicle switch  
{  
    Car {Passengers: 0} => 2.00m + 0.50m,  
    Car {Passengers: 1} => 2.0m,  
    Car {Passengers: 2} => 2.0m - 0.50m,  
    Car                 => 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:

```
C#  
  
vehicle switch  
{  
    // ...  
  
    Taxi {Fares: 0}  => 3.50m + 1.00m,  
    Taxi {Fares: 1}  => 3.50m,  
    Taxi {Fares: 2}  => 3.50m - 0.50m,  
    Taxi             => 3.50m - 1.00m,  
  
    // ...  
};
```

A continuación, implemente las reglas de ocupación mediante la expansión de los casos para los buses, tal como se muestra en el ejemplo siguiente:

```
C#
```

```

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 => 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:

```

C#

vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 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 haya terminado, tendrá un método muy similar al código siguiente:

```

C#

vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car                      => 2.00m - 1.0m,

    Taxi {Fares: 0}   => 3.50m + 1.00m,
    Taxi {Fares: 1}   => 3.50m,
}

```

```

    Taxi {Fares: 2} => 3.50m - 0.50m,
    Taxi           => 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 => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 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, se puede crear un patrón de declaración que se alimenta de un patrón de constante. Esta técnica se muestra en el código siguiente:

```

C#

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 => 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. Puede usar una serie de instrucciones `if` y `else` para expresar esto mediante el código siguiente:

```

C#

public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {

```

```

        return 2.0m;
    }
    else
    {
        return 1.0m;
    }
}
else if (hour < 16)
{
    return 1.5m;
}
else if (hour < 20)
{
    if (inbound)
    {
        return 1.0m;
    }
    else
    {
        return 2.0m;
    }
}
else // Overnight
{
    return 0.75m;
}
}
}
}

```

El código anterior funciona correctamente, pero no es legible. Para que el código tenga sentido, tiene que encadenar todos los casos de entrada y las instrucciones `if` anidadas. En su lugar, usará la coincidencia de patrones para esta característica, 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:

Día	Time	Dirección	Premium
-----	------	-----------	---------

Día	Time	Dirección	Premium
Día de la semana	hora punta de la mañana	hacia la ciudad	x 2,00
Día de la semana	hora punta de la mañana	desde la ciudad	x 1,00
Día de la semana	día	hacia la ciudad	x 1,50
Día de la semana	día	desde la ciudad	x 1,50
Día de la semana	hora punta de la tarde	hacia la ciudad	x 1,00
Día de la semana	hora punta de la tarde	desde la ciudad	x 2,00
Día de la semana	noche	hacia la ciudad	x 0,75
Día de la semana	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:

C#

```
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 es correcto, pero es redundante. Puede simplificarlo tal como se muestra en el código siguiente:

C#

```
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:

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.OVERNIGHT,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _          => TimeBand.EveningRush,
    };
}
```

Agregue un elemento `enum` privado para convertir cada intervalo de tiempo en un valor discreto. Después, el método `GetTimeBand` usa [patrones relacionales](#) y [patrones or conjuntivos](#), ambos agregados en C# 9.0. El patrón relacional permite probar un valor numérico mediante `<`, `>`, `<=` o `>=`. El patrón `or` comprueba si una expresión coincide con uno o más patrones. También puede usar un patrón `and` para asegurarse de que una expresión coincide con dos patrones distintos, y un patrón `not` para probar que una expresión no coincide con un patrón.

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:

C#

```
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:

C#

```
(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:

C#

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

El código debe ser similar al código siguiente después de esos dos cambios:

C#

```

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:

C#

```

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 inaccessible. 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.

Vea también

- [Patrones](#)
- [Expresión switch](#)

Procedimientos para controlar una excepción mediante Try y Catch

Artículo • 07/04/2023

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` que pasa el autor de la llamada.

```
C#  
  
static int GetInt(int[] array, int index)  
{  
    try  
    {  
        return array[index];  
    }  
    catch (IndexOutOfRangeException e) // CS0168  
    {  
        Console.WriteLine(e.Message);  
        // Set IndexOutOfRangeException to the new exception's  
        InnerException.  
        throw new ArgumentException("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 `ArgumentException`, 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 `read-only`, debe asignarla en el constructor de la nueva excepción.

Procedimiento para ejecutar código de limpieza mediante finally

Artículo • 12/03/2024

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:

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./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.

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```
        }
    finally
    {
        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 esté abierto cuando se intente cerrar. El bloque `finally` agrega una comprobación para asegurarse de que el objeto `FileStream` no sea `null` antes de que se 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, es mejor usar el bloque `finally` que esperar a la recolección de elementos no utilizados.

Consulte también

- [using \(Instrucción\)](#)
- [Instrucciones para el control de excepciones](#)



Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

[Proporcionar comentarios sobre el producto](#)

Novedades de C# 13

Artículo • 24/03/2024

C# 13 incluye las siguientes características nuevas. Puede probar estas características con la versión más reciente de [Visual Studio 2022](#) o el [SDK de .NET 9](#).

- Nueva secuencia de escape: `\e`.
- Mejoras de tipo natural de grupo de métodos
- Acceso implícito al indizador en inicializadores de objetos

C# 13 es compatible con .NET 9. Para obtener más información, vea [Control de versiones del lenguaje C#](#).

Puede descargar el SDK de versión preliminar de .NET 9 más reciente desde la [página de descargas de .NET](#). También puede descargar [Visual Studio 2022 en versión preliminar](#), que incluye el SDK de la versión preliminar de .NET 9.

Las nuevas características se agregan a la página "Novedades de C#" cuando están disponibles en versiones preliminares públicas. La sección [conjunto de trabajo](#) de la [página de estado de características de roslyn](#) realiza un seguimiento cuando se combinan las próximas características en la rama principal.

ⓘ Nota

Estamos interesados en sus comentarios sobre estas características. Si encuentra problemas con cualquiera de estas nuevas características, cree un [nuevo problema](#) en el repositorio [dotnet/roslyn](#).

Nueva secuencia de escape

Puede usar `\e` como una secuencia de escape [literal de caracteres](#) para el carácter de **ESCAPE** (U+001B en unicode). Anteriormente, se usaba `\u001b` o `\x1b`. No se recomendaba usar `\x1b` porque si los caracteres que seguían a `1b` eran dígitos hexadecimales válidos, estos caracteres se convertían en parte de la secuencia de escape.

Tipos naturales de grupo de métodos

Esta característica realiza pequeñas optimizaciones para solucionar sobrecargas con grupos de métodos. En el comportamiento anterior, el compilador construía el conjunto

completo de métodos de candidatos para un grupo de métodos. Si se necesitaba un tipo natural, el tipo natural se determinaba a partir del conjunto completo de métodos candidatos.

El nuevo comportamiento consiste en eliminar el conjunto de métodos candidatos en cada ámbito, quitando los métodos candidatos que no son aplicables. Normalmente, estos son métodos genéricos con aridad incorrecta o restricciones que no se cumplen. El proceso continúa hasta el siguiente ámbito externo solo si no se ha encontrado ningún método candidato. Este proceso sigue más detenidamente el algoritmo general para la resolución de sobrecargas. Si todos los métodos candidatos encontrados en un ámbito determinado no coinciden, el grupo de métodos no tiene un tipo natural.

Puede leer los detalles de los cambios en la [especificación de propuesta](#).

Acceso a índices implícitos

El operador de índice implícito "from the end", `^`, ahora se permite en una expresión de inicializador de objeto. Por ejemplo, ahora puede inicializar una matriz en un inicializador de objeto como se muestra en el código siguiente:

C#

```
var v = new S()
{
    buffer =
    {
        [^1] = 0,
        [^2] = 1,
        [^3] = 2,
        [^4] = 3,
        [^5] = 4,
        [^6] = 5,
        [^7] = 6,
        [^8] = 7,
        [^9] = 8,
        [^10] = 9
    }
};
```

En versiones anteriores a C# 13, el operador `^` no se puede usar en un inicializador de objeto. Debe indexar los elementos desde el principio.

Consulte también

- [Novedades de .NET 9](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Novedades de C# 12

Artículo • 19/03/2024

C# 12 incluye las siguientes características nuevas. Puede probar estas características con la versión más reciente de [visual Studio 2022](#) o el [SDK de .NET 8](#).

- [Constructores principales](#): se introdujo en la versión preliminar 2 17.6 de Visual Studio versión 2022.
- [Expresiones de colección](#): introducidas en la versión preliminar 5 17.7 de Visual Studio versión 2022.
- [Matrices insertadas](#): introducidas en la versión preliminar 3 17.7 de Visual Studio versión 2022.
- [Parámetros opcionales en expresiones lambda](#): se introdujo en la versión preliminar 2 17.5 de Visual Studio versión 2022.
- [ref readonly parámetros](#): se introdujo en la versión preliminar 2 17.8 de Visual Studio versión 2022.
- [Alias de cualquier tipo](#): introducido en la versión preliminar 3 17.6 de Visual Studio versión 2022.
- [Atributo experimental](#): se introdujo en la versión preliminar 3 17.7 de Visual Studio, versión 2022.
- [Interceptores - Característica en vista previa \(GB\)](#) introducidas en la versión preliminar 3 17.7 de Visual Studio versión 2022.

C# 12 es compatible con .NET 8. Para obtener más información, vea [Control de versiones del lenguaje C#](#).

Puede descargar el SDK de .NET 8 más reciente de la [página de descargas de .NET](#). También puede descargar [Visual Studio 2022](#), que incluye el SDK de .NET 8.

ⓘ Nota

Estamos interesados en sus comentarios sobre estas características. Si encuentra problemas con cualquiera de estas nuevas características, cree un [nuevo problema](#) en el repositorio [dotnet/roslyn](#).

Constructores principales

Ahora puede crear constructores principales en cualquier `class` y `struct`. Los constructores principales ya no están restringidos a los tipos `record`. Los parámetros del constructor principal están en el ámbito de todo el cuerpo de la clase. Para asegurarse de que se asignan definitivamente todos los parámetros del constructor principal, todos los constructores declarados explícitamente deben llamar al constructor principal mediante la sintaxis `this()`. Agregar un constructor principal a `class` impide que el compilador declare un constructor sin parámetros implícito. En `struct`, el constructor implícito sin parámetros inicializa todos los campos, incluidos los parámetros del constructor principal en el patrón de 0 bits.

El compilador genera propiedades públicas para los parámetros del constructor principal solo en tipos `record`, ya sean tipos `record class` o `record struct`. Es posible que las clases y estructuras que no son de registro no siempre quieran este comportamiento para los parámetros del constructor principal.

Puede obtener más información sobre los constructores principales en el tutorial para [explorar constructores principales](#) y en el artículo sobre [constructores de instancias](#).

Expresiones de colección

Las expresiones de colección presentan una nueva sintaxis terse para crear valores de colección comunes. Insertar otras colecciones en estos valores es posible mediante un operador de propagación `...`.

Se pueden crear varios tipos similares a la colección sin necesidad de compatibilidad con BCL externa. Estos tipos son:

- Tipos de matriz, como `int[]`.
- `System.Span<T>` y `System.ReadOnlySpan<T>`.
- Tipos que admiten inicializadores de colección, como `System.Collections.Generic.List<T>`.

En los ejemplos siguientes se muestran los usos de expresiones de colección:

C#

```
// Create an array:  
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];  
  
// Create a list:  
List<string> b = ["one", "two", "three"];
```

```
// Create a span
Span<char> c = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];

// Create a jagged 2D array:
int[][] twoD = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

// Create a jagged 2D array from variables:
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[][] twoDFromVariables = [row0, row1, row2];
```

El *operador de propagación*, `..` en una expresión de colección reemplaza su argumento por los elementos de esa colección. El argumento debe ser un tipo de colección. En los ejemplos siguientes se muestra cómo funciona el operador de propagación:

C#

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];
foreach (var element in single)
{
    Console.Write($"{element}, ");
}
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

El operando de un operador de propagación es una expresión que se puede enumerar. El operador spread evalúa cada elemento de la expresión de enumeraciones.

Puede usar expresiones de colección en cualquier lugar donde necesite una colección de elementos. Pueden especificar el valor inicial de una colección o pasarse como argumentos a métodos que toman tipos de colección. Puede obtener más información sobre las expresiones de colección en el [artículo de referencia del lenguaje sobre expresiones de colección](#) o la [especificación de características](#).

Parámetros `ref readonly`

C# agregó `in` parámetros como una manera de pasar referencias de solo lectura. Parámetros de `in` permiten variables y valores, y se pueden usar sin ninguna anotación en argumentos.

La adición de parámetros de `ref readonly` permite mayor claridad para las API que podrían usar parámetros de `ref` o parámetros de `in`:

- Las API creadas antes de la introducción de `in` podrían usar `ref` aunque el argumento no se modifique. Esas API se pueden actualizar con `ref readonly`. No será un cambio importante para los autores de llamadas, como sería si el parámetro `ref` se cambiara a `in`. Un ejemplo es [System.Runtime.InteropServices.Marshal.QueryInterface](#).
- Las API que toman un parámetro `in`, pero requieren lógicamente una variable. Una expresión de valor no funciona. Un ejemplo es [System.ReadOnlySpan<T>.ReadOnlySpan<T>\(T\)](#).
- LAS API que usan `ref` porque requieren una variable, pero no mutan esa variable. Un ejemplo es [System.Runtime.CompilerServices.Unsafe.IsNotNullRef](#).

Para obtener más información sobre los parámetros de `ref readonly`, consulte el artículo sobre [los modificadores de parámetros](#) en la referencia del lenguaje o los [parámetros de ref de solo lectura](#) especificación de características.

Parámetros lambda predeterminados

Ahora puede definir valores predeterminados para parámetros en expresiones lambda. La sintaxis y las reglas son las mismas que agregar valores predeterminados para los argumentos a cualquier método o función local.

Puede obtener más información sobre los parámetros predeterminados en las expresiones lambda del artículo sobre [expresiones lambda](#).

Alias de cualquier tipo

Puede usar la directiva de alias `using` para alias de cualquier tipo, no solo de tipos con nombre. Esto significa que puede crear alias semánticos para tipos de tupla, tipos de matriz, tipos de puntero u otros tipos no seguros. Para obtener más información, consulte la [especificación de características](#).

Matrices insertadas

El equipo de runtime usa matrices insertadas y otros autores de bibliotecas para mejorar el rendimiento de las aplicaciones. Las matrices insertadas permiten a un desarrollador crear una matriz de tamaño fijo en un tipo de `struct`. Una estructura con un búfer insertado debe proporcionar características de rendimiento similares a un búfer de

tamaño fijo no seguro. Es probable que no declare sus propias matrices insertadas, pero las usará de forma transparente cuando se exponen como [System.Span<T>](#) o [System.ReadOnlySpan<T>](#) objetos de las API en tiempo de ejecución.

Una *matriz insertada* se declara de forma similar a la siguiente `struct`:

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct Buffer
{
    private int _element0;
}
```

Se usan como cualquier otra matriz:

C#

```
var buffer = new Buffer();
for (int i = 0; i < 10; i++)
{
    buffer[i] = i;
}

foreach (var i in buffer)
{
    Console.WriteLine(i);
}
```

La diferencia es que el compilador puede aprovechar la información conocida sobre una matriz insertada. Es probable que consuma matrices insertadas como lo haría con cualquier otra matriz. Para obtener más información sobre cómo declarar matrices insertadas, consulte la referencia del lenguaje sobre los [tipos de struct](#).

Atributo experimental

Los tipos, métodos o ensamblados se pueden marcar con el [System.Diagnostics.CodeAnalysis.ExperimentalAttribute](#) para indicar una característica experimental. El compilador emite una advertencia si tiene acceso a un método o tipo anotado con el [ExperimentalAttribute](#). Todos los tipos incluidos en un ensamblado marcado con el atributo `Experimental` son experimentales. Puede leer más en el artículo sobre [atributos generales leídos por el compilador](#) o la [especificación de características](#).

Interceptores

Advertencia

Los interceptores son una característica experimental, disponible en modo de vista previa con C# 12. Esta característica puede estar sujeta a cambios importantes o eliminarse en una versión futura. Por lo tanto, no se recomienda para aplicaciones de producción o ya publicadas.

Para usar interceptores, el proyecto de usuario debe especificar la propiedad `<InterceptorsPreviewNamespaces>`. Se trata de una lista de espacios de nombres que pueden contener interceptores.

Por ejemplo:

```
<InterceptorsPreviewNamespaces>$({InterceptorsPreviewNamespaces});Microsoft.AspNetCore.Http.Generated;MyLibrary.Generated</InterceptorsPreviewNamespaces>
```

Un *interceptor* es un método que puede, de forma declarativa, sustituir una llamada a un método *interceptable* por una llamada a sí mismo en tiempo de compilación. Esta sustitución se produce al hacer que el interceptor declare las ubicaciones de origen de las llamadas que intercepta. Los interceptores proporcionan una instalación limitada para cambiar la semántica del código existente agregando código nuevo a una compilación, por ejemplo, en un generador de origen.

Use un *interceptor* como parte de un generador de origen para modificarlo, en lugar de agregar código a una compilación de origen existente. El generador de origen sustituye las llamadas a un método interceptable con una llamada al método *interceptor*.

Si está interesado en experimentar con los interceptores, puede obtener más información en la [especificación de características](#). Si usa la característica, asegúrese de mantenerse al día con los cambios en la especificación de características de esta característica en experimental. Una vez finalizada la característica, agregaremos más instrucciones en este sitio.

Consulte también

- [Novedades de .NET 8](#)



Colaborar con nosotros en
GitHub



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Novedades de C# 11

Artículo • 15/03/2024

Se agregaron las siguientes características en C# 11:

- Literales de cadena sin formato
- Compatibilidad con matemáticas genéricas
- Atributos genéricos
- Literales de cadena UTF-8
- Nuevas líneas en expresiones de interpolación de cadenas
- Patrones de lista
- Tipos locales de archivo
- Miembros requeridos
- Structs predeterminados automáticos
- Coincidencia de patrones de `Span<char>` en una `string` de constante
- Ámbito `nameof` ampliado
- `IntPtr` numérico
- Campos `ref` y `scoped ref`
- Conversión mejorada de grupo de métodos a delegado
- Ola de advertencias 7

C# 11 es compatible con .NET 7. Para obtener más información, vea [Control de versiones del lenguaje C#](#).

Puede descargar el SDK de .NET 7 más reciente de la [página de descargas de .NET](#). También puede descargar [Visual Studio 2022](#), que incluye el SDK de .NET 7.

ⓘ Nota

Estamos interesados en sus comentarios sobre estas características. Si encuentra problemas con cualquiera de estas nuevas características, cree un [nuevo problema](#) en el repositorio [dotnet/roslyn](#).

Atributos genéricos

Puede declarar una [clase genérica](#) cuya clase base sea `System.Attribute`. Esta característica proporciona una sintaxis más cómoda para los atributos que requieren un parámetro `System.Type`. Antes había que crear un atributo que tomara `Type` como parámetro de constructor:

C#

```
// Before C# 11:  
public class TypeAttribute : Attribute  
{  
    public TypeAttribute(Type t) => ParamType = t;  
  
    public Type ParamType { get; }  
}
```

Y, para aplicar el atributo, se usa el operador `typeof`:

C#

```
[TypeAttribute(typeof(string))]  
public string Method() => default;
```

Con esta nueva característica, puede crear un atributo genérico en su lugar:

C#

```
// C# 11 feature:  
public class GenericAttribute<T> : Attribute { }
```

Luego, especifique el parámetro de tipo para usar el atributo:

C#

```
[GenericAttribute<string>()]  
public string Method() => default;
```

Debe proporcionar todos los parámetros de tipo al aplicar el atributo. En otras palabras, el tipo genérico debe [construirse completamente](#). En el ejemplo anterior, los paréntesis vacíos (`()` y `)`) se pueden omitir, ya que el atributo no tiene ningún argumento.

C#

```
public class GenericType<T>  
{  
    [GenericAttribute<T>()] // Not allowed! generic attributes must be fully  
    // constructed types.  
    public string Method() => default;  
}
```

Los argumentos de tipo deben cumplir las mismas restricciones que el operador `typeof`. No se permiten los tipos que requieren anotaciones de metadatos. Por ejemplo, no se

permiten los siguientes tipos como parámetro de tipo:

- `dynamic`
- `string?` (o cualquier tipo de referencia que acepte valores NULL)
- `(int X, int Y)` (o cualquier otro tipo de tupla que use la sintaxis de tupla de C#)

Estos tipos no se representan directamente en los metadatos. Incluyen anotaciones que describen el tipo. En todos los casos, se puede usar el tipo subyacente en su lugar:

- `object` para `dynamic`
- `string` en lugar de `string?`.
- `ValueTuple<int, int>` en lugar de `(int X, int Y)`.

Compatibilidad matemática genérica

Hay varias características de lenguaje que habilitan la compatibilidad matemática genérica:

- `static virtual` miembros en interfaces
- Operador definido por el usuario `checked`
- operadores de desplazamiento relajado
- Operador de desplazamiento a la derecha sin signo

Puede agregar `static abstract` o `static virtual` miembros en interfaces para definir interfaces que incluyan operadores sobrecargables, otros miembros estáticos y propiedades estáticas. El escenario principal de esta característica es usar operadores matemáticos en tipos genéricos. Por ejemplo, puede implementar la interfaz de `System.IAdditionOperators<TSelf, TOther, TResult>` en un tipo que implementa `operator +`. Otras interfaces definen otras operaciones matemáticas o valores bien definidos. Puede obtener información sobre la nueva sintaxis en el artículo sobre [interfaces](#). Las interfaces que incluyen métodos de `static virtual` suelen ser [interfaces genéricas](#). Además, la mayoría declarará una restricción que el parámetro de tipo [implementa en la interfaz declarada](#).

Para más información y probar la característica usted mismo, consulte el tutorial [Exploración de miembros de la interfaz abstracta estática](#) o la entrada de blog [Características en versión preliminar de .NET 6: matemáticas genéricas](#).

Las matemáticas genéricas han creado otros requisitos en el lenguaje.

- *Operador de desplazamiento a la derecha sin signo*: antes de C# 11, para forzar un desplazamiento a la derecha sin signo, había que convertir cualquier tipo entero

con signo en un tipo sin signo, llevar a cabo el desplazamiento y, después, convertir el resultado en un tipo con signo. A partir de C# 11, puede usar `>>>`, el [operador de desplazamiento sin signo](#).

- *Requisitos de operador de desplazamiento menos estrictos:* C# 11 elimina el requisito de que el segundo operando debe ser un `int` o convertible implícitamente en `int`. Este cambio permite que los tipos que implementan interfaces matemáticas genéricas se usen en estas ubicaciones.
- *Operadores definidos por el usuario checked y unchecked:* los desarrolladores ya pueden definir los operadores aritméticos `checked` y `unchecked`. El compilador genera llamadas a la variante correcta en función del contexto actual. Puede obtener más información sobre los `checked` operadores en el artículo sobre [operadores aritméticos](#).

IntPtr y UIntPtr numéricos

Los tipos `nint` y `nuint` ahora se conocen como [System.IntPtr](#) y [System.UIntPtr](#) respectivamente.

Nuevas líneas en interpolaciones de cadenas

El texto dentro de los caracteres `{` y `}` de una interpolación de cadenas ahora puede abarcar varias líneas. El texto entre los marcadores `{` y `}` se analiza como C#. Se permite cualquier C# válido, incluidas las nuevas líneas. Esta característica facilita la lectura de interpolaciones de cadenas que usan expresiones de C# más largas, como expresiones `switch` de coincidencia de patrones o consultas LINQ.

Puede aprender más sobre la característica de nuevas líneas en el artículo [Interpolaciones de cadenas](#) de la referencia del lenguaje.

Patrones de lista

Los *patrones de lista* amplían la coincidencia de patrones para buscar coincidencias con secuencias de elementos de una lista o una matriz. Por ejemplo, `sequence is [1, 2, 3]` es `true` cuando `sequence` es una matriz o una lista de tres enteros (1, 2 y 3). Puede hacer coincidir elementos mediante cualquier patrón, como constante, tipo, propiedad y patrones relacionales. El patrón de descarte (`_`) coincide con cualquier elemento único y el nuevo *patrón de intervalo* (`...`) coincide con cualquier secuencia de cero o más elementos.

Puede encontrar más información sobre los patrones de lista en el artículo sobre [coincidencia de patrones](#) en la referencia del lenguaje.

Conversión mejorada de grupo de métodos a delegado

El estándar de C# en [conversiones de grupo de métodos](#) ahora incluye el siguiente elemento:

- La conversión se permite (pero no necesaria) para usar una instancia de delegado existente que ya contiene estas referencias.

Las versiones anteriores del estándar prohibían al compilador reutilizar el objeto delegado creado para una conversión de grupo de métodos. El compilador de C# 11 almacena en caché el objeto delegado creado a partir de una conversión de grupo de métodos y reutiliza ese objeto delegado único. Esta característica se incluyó por primera vez en Visual Studio 2022 versión 17.2 como característica en vista previa y en .NET 7 (versión preliminar 2).

Literales de cadena sin formato

Los *literales de cadena sin formato* son un nuevo formato para los literales de cadena. Los literales de cadena sin formato pueden contener texto arbitrario, como espacios en blanco, nuevas líneas, comillas insertadas y otros caracteres especiales sin necesidad de secuencias de escape. Un literal de cadena sin formato comienza con al menos tres caracteres de comillas dobles (""""). Y termina con el mismo número de caracteres de comillas dobles. Normalmente, un literal de cadena sin formato usa tres comillas dobles en una sola línea para iniciar la cadena y tres comillas dobles en una línea independiente para finalizar la cadena. Las nuevas líneas que siguen a la comilla de apertura y preceden a la comilla de cierre no se incluyen en el contenido final:

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
            more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
""";
```

Cualquier espacio en blanco a la izquierda de las comillas dobles de cierre se quitará del literal de cadena. Los literales de cadena sin formato se pueden combinar con la interpolación de cadenas para incluir llaves en el texto de salida. Varios caracteres `$` indican cuántas llaves consecutivas comienzan y terminan la interpolación:

```
C#  
  
var location = $$"""  
You are at {{Longitude}}, {{Latitude}}  
""";
```

En el ejemplo anterior se especifica que dos llaves inician y finalizan una interpolación. La tercera llave de apertura y cierre repetida se incluye en la cadena de salida.

Puede encontrar más información sobre los literales de cadena sin formato en el artículo sobre [cadenas de la guía de programación](#) y los artículos de referencia del lenguaje sobre [literales de cadena](#) y [cadenas interpoladas](#).

Estructuras predeterminadas automáticas

El compilador de C# 11 garantiza que todos los campos de un tipo `struct` se inicializarán en su valor predeterminado como parte de la ejecución de un constructor. Este cambio significa que el compilador inicializa automáticamente cualquier campo o propiedad automática no inicializados por un constructor. Las estructuras en las que el constructor no asigna definitivamente todos los campos ahora se compilan, mientras que los campos que no se inicializan explícitamente se establecen en su valor predeterminado. Puede obtener más información sobre cómo afecta este cambio a la inicialización de estructuras en el artículo sobre [estructuras](#).

`Span<char>` de coincidencia de patrón o `ReadOnlySpan<char>` en una constante `string`

Ha podido probar si un elemento `string` tiene un valor constante específico mediante la coincidencia de patrones en varias versiones. Ahora, puede usar la misma lógica de coincidencia de patrones con variables que son `Span<char>` o `ReadOnlySpan<char>`.

Ámbito `nameof` ampliado

Los nombres de parámetro de tipo y los de parámetro ahora están en el ámbito cuando se usan en una expresión `nameof` de una [declaración de atributo](#) del método. Esta

característica significa que puede usar el operador `nameof` para especificar el nombre de un parámetro de método en un atributo de una declaración de método o parámetro. Esta característica suele ser útil para agregar atributos y llevar a cabo [análisis que admitan valores NULL](#).

Literales de cadena de UTF-8

Puede especificar el sufijo `u8` en un literal de cadena para especificar la codificación de caracteres UTF-8. Si la aplicación necesita cadenas UTF-8, para constantes de cadena HTTP o protocolos de texto similares, puede usar esta característica para simplificar la creación de cadenas UTF-8.

Puede obtener más información sobre los literales de cadena UTF-8 en la sección literal de cadena del artículo sobre [los tipos de referencia integrados](#).

Miembros requeridos

Puede agregar el [modificador required](#) a propiedades y campos para aplicar constructores y llamadores para inicializar esos valores. El `System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` se puede agregar a constructores para informar al compilador de que un constructor inicializa *todos los* miembros necesarios.

Para obtener más información sobre los miembros necesarios, consulte la sección [solo inicial](#) del artículo de propiedades.

Campos `ref` y variables `ref scoped`

Puede declarar campos `ref` en [ref struct](#). Esto admite el uso de tipos como `System.Span<T>` sin atributos especiales o tipos internos ocultos.

Puede agregar el modificador `scoped` a cualquier declaración de `ref`. Esto limita el [ámbito](#) al que la referencia puede escapar.

Tipos locales de archivo

A partir de C# 11, el modificador de acceso `file` se puede usar para crear un tipo cuya visibilidad esté limitada al archivo de origen en el que se declara. Esta característica ayuda a los autores de generadores de código fuente a evitar conflictos de

nomenclatura. Puede obtener más información sobre esta característica en el artículo sobre [tipos con ámbito de archivo](#) en la sección de referencia del lenguaje.

Vea también

- [Novedades de .NET 7](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Novedades de C# 10

Artículo • 10/05/2023

C# 10 agrega las características y las mejoras al lenguaje C# siguientes:

- [Structs de registro](#)
- [Mejoras de tipos de estructura](#)
- [Controladores de cadena interpolada](#)
- [Directivas global using](#)
- [Declaración de espacios de nombres con ámbito de archivo](#)
- [Patrones de propiedades extendidos](#)
- [Mejoras en expresiones lambda](#)
- [Se permiten cadenas interpoladas const](#)
- [Los tipos de registro pueden sellar ToString\(\)](#)
- [Asignación definitiva mejorada](#)
- [Se permite la asignación y la declaración en la misma desconstrucción](#)
- [Se permite el atributo AsyncMethodBuilder en los métodos](#)
- [Atributo CallerArgumentExpression](#)
- [Pragma #line mejorado](#)
- [Ola de advertencias 6](#)

C# 10 es compatible con .NET 6. Para obtener más información, vea [Control de versiones del lenguaje C#](#).

Puede descargar el SDK de .NET 6 más reciente de la [página de descargas de .NET](#). También puede descargar [Visual Studio 2022](#), que incluye el SDK de .NET 6.

ⓘ Nota

Estamos interesados en sus comentarios sobre estas características. Si encuentra problemas con cualquiera de estas nuevas características, cree un [nuevo problema](#) en el repositorio [dotnet/roslyn](#).

Structs de registro

Puede declarar los registros de tipo de valor mediante las [declaraciones record struct](#) o [readonly record struct](#). Ahora puede aclarar que un elemento `record` es un tipo de referencia con la declaración `record class`.

Mejoras de tipos de estructura

C# 10 presenta las mejoras siguientes relacionadas con los tipos de estructura:

- Puede declarar un constructor sin parámetros de instancia en un tipo de estructura e inicializar un campo o propiedad de instancia en su declaración. Para más información, consulte la sección [Inicialización de estructuras y valores predeterminados](#) del artículo [Tipos de estructuras](#).
- Un operando izquierdo de la [expresión with](#) puede ser de cualquier tipo de estructura o un tipo anónimo (de referencia).

Controlador de cadena interpolada

Puede crear un tipo que compile la cadena resultante a partir de una [expresión de cadena interpolada](#). Las bibliotecas de .NET usan esta característica en muchas API. Puede compilar una [siguiendo este tutorial](#).

Directivas using globales

Puede agregar el modificador `global` a cualquier [directiva using](#) para indicar al compilador que la directiva se aplica a todos los archivos de código fuente de la compilación. Normalmente, se trata de todos los archivos de código fuente de un proyecto.

Declaración de espacios de nombres con ámbito de archivo

Puede usar una nueva forma de la [declaración namespace](#) para declarar que todas las declaraciones posteriores son miembros del espacio de nombres declarado:

C#

```
namespace MyNamespace;
```

Esta nueva sintaxis ahorra espacio horizontal y vertical para las declaraciones `namespace`.

Patrones de propiedades extendidos

A partir de C# 10, puede hacer referencia a propiedades o campos anidados en un patrón de propiedad. Por ejemplo, un patrón con el formato

```
C#  
{ Prop1.Prop2: pattern }
```

es válido en C# 10 y versiones posteriores, y equivalente a

```
C#  
{ Prop1: { Prop2: pattern } }
```

válido en C# 8.0 y versiones posteriores.

Para obtener más información, consulte la nota de propuesta de características [Patrones de propiedades extendidos](#). Para obtener más información sobre un patrón de propiedades, vea la sección [Patrón de propiedad](#) del artículo [Patrones](#).

Mejoras de expresiones lambda

C# 10 incluye muchas mejoras en el modo en que se controlan las expresiones lambda:

- Las expresiones lambda pueden tener un [tipo natural](#), donde el compilador puede deducir un tipo delegado de la expresión lambda o del grupo de métodos.
- Las expresiones lambda pueden declarar un [tipo de valor devuelto](#) cuando el compilador no puede deducirlo.
- Los [atributos](#) se pueden aplicar a las expresiones lambda.

Estas características hacen que las expresiones lambda sean parecidas a los métodos y las funciones locales. Facilitan el uso de expresiones lambda sin declarar una variable de un tipo delegado y funcionan de forma más fluida con las nuevas API mínimas de ASP.NET CORE.

Cadenas interpoladas constantes

En C# 10, las cadenas `const` se pueden inicializar mediante la [interpolación de cadenas](#) si todos los marcadores de posición son cadenas constantes. La interpolación de cadenas puede crear cadenas constantes más legibles a medida que se compilan las cadenas constantes usadas en la aplicación. Las expresiones de marcador de posición no pueden ser constantes numéricas porque esas constantes se convierten en cadenas en

tiempo de ejecución. La referencia cultural actual puede afectar a su representación de cadena. Obtenga más información en la referencia del lenguaje sobre [expresiones const.](#)

Los tipos de registro pueden sellar ToString

En C# 10, puede agregar el modificador `sealed` al invalidar `ToString` en un tipo de registro. Al sellar el método `ToString`, se impide que el compilador sintetice un método `ToString` para cualquier tipo de registro derivado. Un elemento `ToString sealed` (sellado) garantiza que todos los tipos de registros derivados usen el método `ToString`, definido en un tipo de registro base común. Puede obtener más información sobre esta característica en el artículo sobre [registros](#).

Asignación y declaración en la misma desconstrucción

Este cambio quita una restricción de versiones anteriores de C#. Anteriormente, una desconstrucción podía asignar todos los valores a variables existentes, o bien inicializar variables recién declaradas:

```
C#  
  
// Initialization:  
(int x, int y) = point;  
  
// assignment:  
int x1 = 0;  
int y1 = 0;  
(x1, y1) = point;
```

En C# 10 se elimina esta restricción:

```
C#  
  
int x = 0;  
(x, int y) = point;
```

Asignación definitiva mejorada

Antes de C# 10, había muchos escenarios en los que la asignación definitiva y el análisis de estado NULL generaban advertencias que eran falsos positivos. Por lo general, estas implicaban comparaciones con constantes booleanas, el acceso a una variable solo en

las instrucciones `true` o `false` de una instrucción `if`, y expresiones de uso combinado de `NULL`. Estos ejemplos generaron advertencias en versiones anteriores de C#, pero no en C# 10:

C#

```
string representation = "N/A";
if ((c != null && c.GetDependentValue(out object obj)) == true)
{
    representation = obj.ToString(); // undesired error
}

// Or, using ?.
if (c?.GetDependentValue(out object obj) == true)
{
    representation = obj.ToString(); // undesired error
}

// Or, using ??
if (c?.GetDependentValue(out object obj) ?? false)
{
    representation = obj.ToString(); // undesired error
}
```

El impacto principal de esta mejora es que las advertencias para la asignación definitiva y el análisis de estado `NULL` son más precisas.

Se permite el atributo `AsyncMethodBuilder` en los métodos

En C# 10 y versiones posteriores, puede especificar un generador de métodos asíncronos diferente para un único método, además de especificar el tipo de generador de métodos para todos los métodos que devuelven un tipo concreto similar a una tarea. Un generador de métodos asíncronos personalizado permite escenarios avanzados de optimización del rendimiento en los que un método determinado puede beneficiarse de un generador personalizado.

Para obtener más información, vea la sección sobre `AsyncMethodBuilder` del artículo sobre los atributos leídos por el compilador.

Diagnóstico del atributo `CallerArgumentExpression`

Puede usar `System.Runtime.CompilerServices.CallerArgumentExpressionAttribute` para especificar un parámetro que el compilador reemplace por la representación de texto de otro argumento. Esta característica permite a las bibliotecas crear diagnósticos más específicos. El código siguiente prueba una condición. Si la condición es falsa, el mensaje de excepción incluye la representación de texto del argumento pasado a `condition`:

C#

```
public static void Validate(bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation:
<{message}>");
    }
}
```

Puede obtener más información sobre esta característica en el artículo sobre [Atributos de información del autor de la llamada](#) en la sección de referencia del lenguaje.

Pragma #line mejorado

C# 10 admite un formato nuevo para el pragma `#line`. Es probable que no use el formato nuevo, pero verá sus efectos. Las mejoras permiten una salida más detallada en lenguajes específicos de dominio (DSL), como Razor. El motor de Razor usa estas mejoras para mejorar la experiencia de depuración. Encontrará que los depuradores pueden resaltar el origen de Razor con más precisión. Para obtener más información sobre la nueva sintaxis, vea el artículo sobre [Directivas de preprocesador](#) en la referencia del lenguaje. También puede leer la [especificación de la característica](#) para obtener ejemplos basados en Razor.

Conozca los cambios más importantes en el compilador de C#.

Artículo • 10/05/2023

Puede encontrar cambios importantes desde la versión de C# 10 [aquí](#).

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 Roslyn en C# 10.0/.NET 6](#)
- [Cambios importantes en Roslyn después de .NET 5](#)
- [Cambios importantes en la versión 16.8 de VS2019 incorporados con .NET 5 y C# 9.0](#)
- [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# \(VS2013 y anterior\).](#)
- [Cambio de versión de Unicode en C# 6](#)

Historia de C#

Artículo • 17/04/2024

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.

ⓘ Importante

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.

C# versión 12

Publicado en noviembre de 2023

En C# 12 se han agregado las siguientes características:

- [Constructores principales](#): puede crear constructores principales en cualquier tipo de `class` o `struct`.
- [Expresiones de colección](#): una nueva sintaxis para especificar expresiones de colección, incluido el operador de propagación, (`...`), para expandir cualquier colección.
- [Matrices insertadas](#): las matrices insertadas permiten crear una matriz de tamaño fijo en un tipo de `struct`.
- [Parámetros opcionales en expresiones lambda](#): puede definir valores predeterminados para parámetros en expresiones lambda.

- [ref readonly parámetros](#) - `ref readonly` permite mayor claridad para las API que podrían usar parámetros `ref` o parámetros `in`.
- [Alias de cualquier tipo](#): puede usar la directiva de alias `using` para alias de cualquier tipo, no solo de tipos con nombre.
- [Atributo experimental](#): indique una característica experimental.

Además, [Interceptores](#): se lanzó como una *característica en vista previa*.

En general, C# 12 proporciona nuevas características que hacen que sea más productivo escribir código de C#. La sintaxis que ya sabía está disponible en más sitios. Otra sintaxis permite la coherencia de los conceptos relacionados.

C# versión 11

Fecha de publicación noviembre de 2022

Se agregaron las siguientes características en C# 11:

- [Literales de cadena sin formato](#)
- [Compatibilidad con matemáticas genéricas](#)
- [Atributos genéricos](#)
- [Literales de cadena UTF-8](#)
- [Nuevas líneas en expresiones de interpolación de cadenas](#)
- [Patrones de lista](#)
- [Tipos locales de archivo](#)
- [Miembros requeridos](#)
- [Structs predeterminados automáticos](#)
- [Coincidencia de patrones de Span<char> en una string de constante](#)
- [Ámbito nameof ampliado](#)
- [IntPtr numérico](#)
- [Campos ref y scoped ref](#)
- [Conversión mejorada de grupo de métodos a delegado](#)
- [Ola de advertencias 7](#)

C# 11 presenta *matemáticas genéricas* y varias características que admiten ese objetivo. Puede escribir algoritmos numéricos una vez para todos los tipos de números. Hay más características para facilitar el trabajo con `struct` tipos, como los miembros necesarios y las estructuras predeterminadas automáticas. Trabajar con cadenas resulta más fácil con literales de cadena sin formato, nueva línea en interpolaciones de cadenas y literales de cadena UTF-8. Las características como los tipos locales de archivo permiten que los generadores de origen sean más sencillos. Por último, los patrones de lista agregan más compatibilidad con la coincidencia de patrones.

C# versión 10

Fecha de publicación noviembre de 2021

C# 10 agrega las características y las mejoras al lenguaje C# siguientes:

- Structs de registro
- Mejoras de tipos de estructura
- Controladores de cadena interpolada
- Directivas global using
- Declaración de espacios de nombres con ámbito de archivo
- Patrones de propiedades extendidos
- Mejoras en expresiones lambda
- Se permiten cadenas interpoladas const
- Los tipos de registro pueden sellar ToString()
- Asignación definitiva mejorada
- Se permite la asignación y la declaración en la misma desconstrucción
- Se permite el atributo AsyncMethodBuilder en los métodos
- Atributo CallerArgumentExpression
- Pragma #line mejorado

Hay más características disponibles en el modo *de vista previa*. Para poder usar estas características, debe establecer <LangVersion> en Preview en el proyecto:

- Los [atributos genéricos](#) se explican más adelante en este artículo.
- [Miembros abstractos estáticos en interfaces](#).

C# 10 sigue trabajando en temas de eliminación de ceremonias, separación de datos de los algoritmos y rendimiento mejorado para .NET Runtime.

Muchas de las características le permiten escribir menos código para expresar los mismos conceptos. Los *structs de registro* sintetizan muchos de los mismos métodos que las *clases de registro*. Los structs y los tipos anónimos son *compatibles con las expresiones*. Las *directivas de uso global* y las *declaraciones de espacio de nombres con ámbito de archivo* consiguen que exprese las dependencias y la organización del espacio de nombres con mayor claridad. *Las mejoras de lambda* facilitan la declaración de expresiones lambda en las que se usan. Los nuevos patrones de propiedad y las mejoras de deconstrucción crean un código más conciso.

Los nuevos controladores de cadenas interpoladas y el comportamiento `AsyncMethodBuilder` pueden mejorar el rendimiento. Estas características de lenguaje se aplicaron en .NET Runtime para lograr mejoras de rendimiento en .NET 6.

C# 10 también supone un cambio más profundo en la cadencia anual de los lanzamientos de .NET. Dado que no todas las características se pueden completar en un período anual, puede probar un par de características en "versión preliminar" de C# 10. Se pueden usar *atributos genéricos* y *miembros abstractos estáticos en interfaces*, pero son características en versión preliminar y pueden cambiar antes de su versión final.

C# versión 9

Fecha de publicación noviembre de 2020

C# 9 se publicó con .NET 5. Es la versión de lenguaje predeterminada para cualquier ensamblado que tenga como destino la versión de .NET 5. Contiene las siguientes características nuevas y mejoradas:

- [Registros](#)
- [Establecedores de solo inicialización](#)
- [Instrucciones de nivel superior](#)
- Mejoras de coincidencia de patrones: [patrones relacionales](#) y [patrones lógicos](#)
- [Rendimiento e interoperabilidad](#)
 - [Enteros con tamaño nativos](#)
 - [Punteros de función](#)
 - [Supresión de la emisión de la marca localsinit](#)
 - [Inicializadores de módulo](#)
 - [Nuevas características para métodos parciales](#)
- [Características de ajuste y finalización](#)
 - [Expresiones new con tipo de destino](#)
 - [Funciones anónimas static](#)
 - [Expresiones condicionales con tipo de destino](#)
 - [Tipos de valor devueltos de covariante](#)
 - [Compatibilidad con extensiones GetEnumerator para bucles foreach](#)
 - [Parámetros de descarte lambda](#)
 - [Atributos en funciones locales](#)

C# 9 continúa tres de los temas de versiones anteriores: quitar complejidad, separar datos de algoritmos y proporcionar más patrones en más lugares.

Las [instrucciones de nivel superior](#) hacen que el programa principal sea más fácil de leer. La complejidad es innecesaria: un espacio de nombres, una clase `Program` y `static void Main()` son innecesarios.

La introducción de [records](#) proporciona una sintaxis concisa para los tipos de referencia que siguen la semántica del valor para la igualdad. Usará estos tipos para definir

contenedores de datos que normalmente definen un comportamiento mínimo. Los establecimientos de solo inicialización proporcionan la funcionalidad para la mutación no destructiva `with` (expresiones) en los registros. C# 9 también agrega tipos de valor devuelto de covariante para que los registros derivados puedan invalidar los métodos virtuales y devolver un tipo derivado del tipo de valor devuelto del método base.

Las funcionalidades de coincidencia de patrones se expanden de varias maneras. Los tipos numéricos ahora admiten *patrones de rango*. Los patrones se pueden combinar mediante los patrones `and`, `or` y `not`. Se pueden agregar paréntesis para aclarar patrones más complejos:

C# 9 incluye nuevas mejoras de coincidencia de patrones:

- Los **patrones de tipo** coinciden con un objeto que coincide con un tipo determinado.
- Los **patrones con paréntesis** aplican o resaltan la prioridad de las combinaciones de patrones
- En los **patrones and conjuntivos** es necesario que los dos patrones coincidan.
- En los **patrones or disyuntivos** es necesario que alguno de los patrones coincida
- En los **patrones not negados** es necesario que un patrón no coincida.
- Los **patrones relacionales** requieren que la entrada sea menor que, mayor que, menor o igual que, o mayor o igual que una constante determinada

Estos patrones enriquecen la sintaxis de los patrones. Tenga en cuenta estos ejemplos:

C#

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Con paréntesis opcionales para que quede claro que `and` tiene mayor prioridad que `or`:

C#

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

Uno de los usos más comunes es una nueva sintaxis para una comprobación NULL:

C#

```
if (e is not null)
{
```

```
// ...  
}
```

Cualquiera de estos patrones se puede usar en cualquier contexto en el que se permitan patrones: expresiones de patrón `is`, expresiones `switch`, patrones anidados y el patrón de la etiqueta `case` de una instrucción `switch`.

Otro conjunto de características admite la informática de alto rendimiento en C#:

- Los tipos `nint` y `nuint` modelan los tipos enteros de tamaño nativo en la CPU de destino.
- Los [punteros de función](#) proporcionan una funcionalidad similar a la de un delegado, al mismo tiempo que evitan las asignaciones necesarias para crear un objeto delegado.
- La instrucción `localsinit` se puede omitir para guardar instrucciones.

Rendimiento e interoperabilidad

Otro conjunto de mejoras admite escenarios en los que los *generadores de código* agregan funcionalidad:

- Los [inicializadores de módulo](#) son métodos a los que el runtime llama cuando se carga un ensamblado.
- Los [métodos parciales](#) admiten nuevos modificadores de acceso y tipos de valor devuelto distintos de void. En esos casos, se debe proporcionar una implementación.

Características de ajuste y finalización

C# 9 agrega muchas otras pequeñas características que mejoran la productividad del desarrollador, tanto para escribir como para leer código:

- Expresiones `new` con tipo de destino
- Funciones anónimas `static`
- Expresiones condicionales con tipo de destino
- Compatibilidad con extensiones `GetEnumerator()` para bucles `foreach`
- Las expresiones lambda pueden declarar parámetros de descarte
- Los atributos se pueden aplicar a las funciones locales

La versión C# 9 continúa el trabajo para hacer que C# siga siendo un lenguaje de programación moderno y de uso general. Las características siguen siendo compatibles con cargas de trabajo y tipos de aplicación modernos.

C# versión 8.0

Fecha de publicación septiembre de 2019

C# 8.0 es la primera versión C# principal que tiene como destino específicamente .NET Core. Algunas características se basan en nuevas funcionalidades de Common Language Runtime (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.

C# versión 7.3

Fecha de publicación mayo de 2018

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. También 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 más genéricas.

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 adjuntar atributos al campo de respaldo de propiedades implementadas automáticamente.
- Se ha mejorado la resolución de métodos cuando los argumentos difieren en `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.

C# versión 7.2

Fecha de publicación noviembre de 2017

C#7.2 agregó varias características de lenguaje pequeñas:

- Inicializadores en matrices `stackalloc`.
- Uso de instrucciones `fixed` con cualquier tipo que admita un patrón.
- Acceso campos fijos sin anclar.
- Reasignación de variables locales `ref`.
- Declaración de tipos `readonly struct` para indicar que una estructura es fija y que debería pasarse como parámetro `in` a los métodos de su miembro.
- Incorporación del 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.

- Uso del 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.
- Declaración de tipos `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.
- Uso de restricciones genéricas adicionales.
- **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 condicionales `ref`:
 - El resultado de una expresión condicional (`? :`) ahora puede ser una referencia.

C# versión 7.1

Fecha de publicación agosto de 2017

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:

- **asyncMain method**
 - 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 7.0

Fecha de publicación marzo de 2017

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. Estas son algunas de las nuevas características:

- Variables out
- [Tuplas y deconstrucción](#)
- [Coincidencia de patrones](#)
- [Funciones locales](#)
- [Miembros con forma de expresión expandidos](#)
- [Variables locales de tipo ref](#)
- [Devoluciones de referencias](#)

Otras características incluidas:

- [Descartes](#)
- [Literales binarios y separadores de dígitos](#)
- [Expresiones throw](#)

Todas estas características ofrecen capacidades nuevas 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. .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 6.0

Fecha de publicación julio de 2015

Versión 6.0, publicada con Visual Studio 2015, lanzó muchas características más pequeñas que hicieron que la programación de C# sea más productiva. 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

Entre las otras características nuevas se incluyen estas:

- Inicializadores de índice
- Await en bloques catch y finally
- Valores predeterminados para las propiedades solo de captador

Si observa estas características juntas, verá un patrón interesante. En esta versión, C# comenzó a eliminar el lenguaje reutilizable para que el código sea más breve y legible. 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 5.0

Fecha de publicación agosto de 2012

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
- 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.

C# versión 4.0

Fecha de publicación abril de 2010

La versión 4.0 de C#, publicada con Visual Studio 2010, introdujo algunas características interesantes:

- Enlace dinámico
- Argumentos opcionales/con nombre
- Covariante y contravariante de genéricos
- Tipos de interoperabilidad insertados

Los tipos de interoperabilidad incrustados facilitaron el problema de implementación de crear ensamblados de interoperabilidad COM para la aplicación. 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 3.0

Fecha de publicación noviembre de 2007

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. La mejor característica de la versión de C# era 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 comenzó 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 fuera mucho más inteligente.

C# versión 2.0

Fecha de publicación noviembre de 2005

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 C# comenzase como un lenguaje genérico orientado a objetos, la versión 2.0 de C# lo cambió enseguida. 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. Es mejor usar genéricos que crear un tipo `ListInt` que derive de `ArrayList` o convertir 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.

Versión 1.2 de C#

Fecha de publicación abril de 2003

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 1.0

Fecha de publicación enero de 2002

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 declarados para ECMA](#), buscaba ser un “lenguaje sencillo, moderno, orientado a objetos y de uso general”. En aquella época, parecerse a Java suponía conseguir 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
- Operadores y expresiones
- Instrucciones
- Atributos

Artículo [publicado originalmente en el blog de NDepend](#), por cortesía de Erik Dietrich y Patrick Smacchia.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Relaciones entre características de lenguaje y tipos de biblioteca

Artículo • 08/11/2023

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. Por este motivo, las versiones de C# solo se admiten para la versión correspondiente de .NET y versiones más recientes. Esto garantiza el comportamiento correcto en tiempo de ejecución y la disponibilidad de todos los tipos y miembros necesarios.

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](#) : se usa para todas las excepciones generadas por el compilador.
- [String](#): 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. Algunos ejemplos son: [INotifyCompletion](#), [CallerFilePathAttribute](#) y [CallerMemberNameAttribute](#). C# 7.0 agregó una dependencia de [ValueTuple](#) para implementar la característica de lenguaje [tuplas](#). C# 8 requiere [System.Index](#) y [System.Range](#) para [intervalos e índices](#), entre otras características. Cada nueva versión puede agregar requisitos adicionales.

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. Las herramientas del compilador de C# ahora se han desvinculado del ciclo de versiones de las bibliotecas de .NET en las plataformas compatibles.



Colaborar con nosotros en



.NET feedback

GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

The .NET documentation is open source. Provide feedback [here](#).

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Consideraciones sobre versiones y actualizaciones para desarrolladores de C#

Artículo • 03/07/2023

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. El equipo del entorno de ejecución de .NET también tiene previsto garantizar la compatibilidad con las bibliotecas actualizadas. En casi todos los casos, cuando se inicia la aplicación desde un entorno de ejecución con bibliotecas actualizadas, el comportamiento es exactamente el mismo que con las versiones anteriores.

La versión del lenguaje que se usa para compilar la aplicación suele coincidir con el moniker de la plataforma de destino en tiempo de ejecución (TFM) al que se hace referencia en el proyecto. Para obtener más información sobre cómo cambiar la versión predeterminada del lenguaje, consulte el artículo [Control de versiones del lenguaje C#](#). Este comportamiento predeterminado garantiza la máxima compatibilidad.

Cuando se introducen *cambios importantes*, se clasifican del siguiente modo:

- *Cambio importante de código binario*: un cambio importante de código binario provoca un comportamiento diferente, incluidos posibles bloqueos, en la aplicación o biblioteca cuando se inicia con un nuevo entorno de ejecución. Debe volver a compilar la aplicación para incorporar estos cambios. El código binario anterior no funcionará correctamente.
- *Cambio importante de código fuente*: un cambio importante en el código fuente cambia su significado. Debe modificar el código fuente antes de compilar la aplicación con la versión más reciente del lenguaje. El código binario anterior se ejecutará correctamente con el host y el entorno de ejecución más recientes. Tenga en cuenta que, para la sintaxis del lenguaje, un *cambio importante de código fuente* también es un *cambio de comportamiento*, como se define en los [cambios importantes del entorno de ejecución](#).

Cuando un cambio importante de código binario afecta a la aplicación, debe volver a compilar la aplicación, pero no es necesario editar ningún código fuente. Cuando un cambio importante de código fuente afecta a la aplicación, el código binario anterior se sigue ejecutando correctamente en sistemas que tienen el entorno de ejecución y las bibliotecas actualizados. Sin embargo, debe realizar cambios en el código fuente para volver a compilar la aplicación con la nueva versión del lenguaje y el entorno de

ejecución. En el caso de un cambio importante tanto de código binario como de código fuente, debe volver a compilar la aplicación con la versión más reciente y actualizar el código fuente.

Debido al objetivo que tienen los equipos del lenguaje C# y del entorno de ejecución de evitar cambios importantes, la actualización de la aplicación suele ser una cuestión de actualizar el TFM y recompilar la aplicación. Sin embargo, para las bibliotecas que se distribuyen públicamente, debe evaluar cuidadosamente su directiva de TFM admitidos y las versiones de lenguaje admitidas. 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 con la posibilidad de que muchos de los usuarios aún no tengan versiones actualizadas.

Introducción de cambios importantes en bibliotecas

Cuando adopte nuevas características de lenguaje en la API pública de la biblioteca, debe evaluar si la adopción de la característica presenta un cambio importante de código binario o de código fuente para los usuarios de la biblioteca. Los cambios en la implementación interna que no aparecen en las interfaces `public` o `protected` son compatibles.

ⓘ Nota

Si usa `System.Runtime.CompilerServices.InternalsVisibleToAttribute` para permitir que los tipos muestren los miembros internos, estos miembros internos pueden introducir cambios importantes.

Un *cambio importante de código binario* requiere que los usuarios vuelvan a compilar el código para usar la nueva versión. Por ejemplo, considere este método público:

C#

```
public double CalculateSquare(double value) => value * value;
```

Si agrega el modificador `in` al método, se trata de un cambio importante de código binario:

C#

```
public double CalculateSquare(in double value) => value * value;
```

Los usuarios deben volver a compilar cualquier aplicación que use el método `CalculateSquare` para que la nueva biblioteca funcione correctamente.

Un *cambio importante de código fuente* requiere que los usuarios cambien el código antes de volver a compilar la aplicación. Por ejemplo, considere este tipo:

C#

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);

    // other details omitted
}
```

En una versión más reciente, le gustaría aprovechar las ventajas de los miembros sintetizados generados para los tipos `record`. Realice el siguiente cambio:

C#

```
public record class Person(string FirstName, string LastName);
```

El cambio anterior requiere modificaciones en cualquier tipo derivado de `Person`. Todas esas declaraciones deben agregar el modificador `record` a sus declaraciones.

Impacto de los cambios importantes

Al agregar un *cambio importante de código binario* a la biblioteca, se obliga a que todos los proyectos que usen la biblioteca se vuelvan a compilar. Sin embargo, no es necesario modificar el código fuente de esos proyectos. Por tanto, el impacto del cambio importante es razonablemente pequeño para cada proyecto.

Al realizar un *cambio importante de código fuente* en la biblioteca, es necesario cambiar el código fuente de todos los proyectos para que usen la nueva biblioteca. Si el cambio necesario requiere nuevas características del lenguaje, obliga a actualizar esos proyectos a la misma versión del lenguaje y el TFM que usa ahora. Ha requerido más trabajo para los usuarios y, posiblemente, también los ha obligado a actualizar la aplicación.

El impacto de cualquier cambio importante que realice depende del número de proyectos que tengan una dependencia de la biblioteca. Si algunas aplicaciones internas usan la biblioteca, puede reaccionar ante cualquier cambio importante en todos los proyectos afectados. Sin embargo, si la biblioteca se descarga públicamente, debe evaluar el posible impacto y considerar alternativas:

- Agregar nuevas API que funcionen en paralelo con las API actuales.
- Compilaciones paralelas para diferentes TFM.
- Compatibilidad con múltiples versiones.

Tutorial: Exploración de constructores principales

Artículo • 28/06/2023

C# 12 presenta *constructores principales*, una sintaxis concisa para declarar constructores cuyos parámetros están disponibles en cualquier parte del cuerpo del tipo.

En este tutorial, aprenderá a:

- ✓ Cuándo declarar un constructor principal en el tipo
- ✓ Cómo llamar a constructores principales desde otros constructores
- ✓ Uso de parámetros de constructor principal en miembros del tipo
- ✓ Dónde se almacenan los parámetros del constructor principal

Requisitos previos

Tendrá que configurar la máquina para ejecutar .NET 8 o posterior, incluido el compilador de C# 12 o posterior. El compilador de C# 12 está disponible a partir de la [versión 17.7 de Visual Studio 2022](#) o del [SDK de .NET 8](#).

Constructores principales

Puede agregar parámetros a una declaración `struct` o `class` para crear un *constructor principal*. Los parámetros del constructor principal están en el ámbito en toda la definición de clase. Es importante ver los parámetros del constructor principal como *parámetros* aunque estén en el ámbito a lo largo de la definición de clase. Varias reglas aclaran que son parámetros:

1. Es posible que los parámetros del constructor principal no se almacenen si no son necesarios.
2. Los parámetros del constructor principal no son miembros de la clase. Por ejemplo, no se puede tener acceso a un parámetro de constructor principal denominado `param` como `this.param`.
3. Se pueden asignar parámetros de constructor principal.
4. Los parámetros del constructor principal no se convierten en propiedades, excepto en los tipos [record](#).

Estas reglas son las mismas que los parámetros de cualquier método, incluidas otras declaraciones de constructor.

Los usos más habituales para un parámetro de constructor principal son:

1. Como argumento para una invocación de constructor `base()`.
2. Para inicializar un campo o propiedad miembro.
3. Hacer referencia al parámetro constructor en un miembro de instancia.

Todos los demás constructores de una clase **deben** llamar al constructor principal, directa o indirectamente, a través de una invocación de constructor `this()`. Esta regla garantiza que los parámetros del constructor principal se asignen en cualquier lugar del cuerpo del tipo.

Inicializar propiedad

El código siguiente inicializa dos propiedades de solo lectura que se calculan a partir de los parámetros del constructor principal:

```
C#  
  
public readonly struct Distance(double dx, double dy)  
{  
    public readonly double Magnitude { get; } = Math.Sqrt(dx * dx + dy *  
dy);  
    public readonly double Direction { get; } = Math.Atan2(dy, dx);  
}
```

El código anterior muestra un constructor principal que se usa para inicializar las propiedades de solo lectura calculadas. Inicializadores de campo para `Magnitude` y `Direction` usan los parámetros del constructor principal. Los parámetros del constructor principal no se usan en ningún otro lugar de la estructura. La estructura anterior es como si hubiera escrito el código siguiente:

```
C#  
  
public readonly struct Distance  
{  
    public readonly double Magnitude { get; }  
  
    public readonly double Direction { get; }  
  
    public Distance(double dx, double dy)  
    {  
        Magnitude = Math.Sqrt(dx * dx + dy * dy);  
        Direction = Math.Atan2(dy, dx);  
    }  
}
```

La nueva característica facilita el uso de inicializadores de campo cuando se necesitan argumentos para inicializar un campo o una propiedad.

Creación de un estado mutable

En los ejemplos anteriores se usan parámetros de constructor principal para inicializar propiedades de solo lectura. También puede usar constructores principales cuando las propiedades no son de solo lectura. Observe el código siguiente:

C#

```
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

En el ejemplo anterior, el método `Translate` cambia los componentes `dx` y `dy`. Esto requiere que las propiedades `Magnitude` y `Direction` se calculen al acceder a ellas. El operador `=>` designa un descriptor de acceso `get` con forma de expresión, mientras que el operador `=` designa un inicializador. Esta versión agrega un constructor sin parámetros a la estructura. El constructor sin parámetros debe invocar al constructor principal, de modo que se inicialicen todos los parámetros del constructor principal.

En el ejemplo anterior, se obtiene acceso a las propiedades del constructor principal en un método. Por lo tanto, el compilador crea campos ocultos para representar cada parámetro. El código siguiente muestra aproximadamente lo que genera el compilador. Los nombres de campo reales son identificadores CIL válidos, pero no identificadores de C# válidos.

C#

```
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
```

```

        __unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
    }

    public Distance(double dx, double dy)
    {
        __unspeakable_dx = dx;
        __unspeakable_dy = dy;
    }
    public Distance() : this(0, 0) { }
}

```

Es importante comprender que el primer ejemplo no requería que el compilador creara un campo para almacenar el valor de los parámetros del constructor principal. En el segundo ejemplo se ha usado el parámetro de constructor principal dentro de un método y, por tanto, ha sido necesario que el compilador creara almacenamiento para ellos. El compilador crea almacenamiento para cualquier constructor principal solo cuando se accede a ese parámetro en el cuerpo de un miembro del tipo. De lo contrario, los parámetros del constructor principal no se almacenan en el objeto.

Inserción de dependencias

Otro uso habitual de los constructores principales es especificar parámetros para la inserción de dependencias. El código siguiente crea un controlador simple que requiere una interfaz de servicio para su uso:

C#

```

public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}

```

El constructor principal indica claramente los parámetros necesarios en la clase. Los parámetros del constructor principal se usan como haría con cualquier otra variable de la clase.

Inicializar clase base

Puede invocar el constructor principal de una clase base desde el constructor principal de la clase derivada. Es la manera más fácil de escribir una clase derivada que debe invocar un constructor principal en la clase base. Por ejemplo, considere una jerarquía de clases que representan diferentes tipos de cuenta como banco. La clase base tendría un aspecto similar al código siguiente:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner: {Owner}";
}
```

Todas las cuentas bancarias, independientemente del tipo, tienen propiedades para el número de cuenta y un propietario. En la aplicación completada, se agregaría otra funcionalidad común a la clase base.

Muchos tipos requieren una validación más específica en los parámetros del constructor. Por ejemplo, `BankAccount` tiene requisitos específicos para los parámetros `owner` y `accountID`: `owner` no puede ser `null` ni espacio en blanco, y `accountID` debe ser una cadena que contenga 10 dígitos. Puede agregar esta validación al asignar las propiedades correspondientes:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number",
            nameof(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
            nameof(owner))
        : owner;
```

```
    public override string ToString() => $"Account ID: {AccountID}, Owner: {Owner}";

    public static bool ValidAccountNumber(string accountID) =>
        accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

En el ejemplo anterior se muestra cómo se pueden validar los parámetros del constructor antes de asignarlos a las propiedades. Puede usar métodos integrados, como `String.IsNullOrEmpty(String)`, o su propio método de validación, como `ValidAccountNumber`. En el ejemplo anterior, las excepciones se inician desde el constructor cuando invoca los inicializadores. Si no se usa un parámetro de constructor para asignar un campo, se inician excepciones cuando se accede por primera vez al parámetro constructor.

Una clase derivada presentaría una cuenta corriente:

```
C#

public class CheckingAccount(string accountID, string owner, decimal overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds for withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID}, Owner: {Owner}";
```

```
{Owner}, Balance: {CurrentBalance}";  
}
```

La clase derivada `CheckingAccount` tiene un constructor principal que toma todos los parámetros necesarios en la clase base y otro parámetro con un valor predeterminado. El constructor principal llama al constructor base mediante la sintaxis : `BankAccount(accountID, owner)`. Esta expresión especifica tanto el tipo de la clase base como los argumentos del constructor principal.

La clase derivada no es necesaria para usar un constructor principal. Puede crear un constructor en la clase derivada que invoque el constructor principal de la clase base, como se muestra en el ejemplo siguiente:

C#

```
public class LineOfCreditAccount : BankAccount  
{  
    private readonly decimal _creditLimit;  
    public LineOfCreditAccount(string accountID, string owner, decimal creditLimit) : base(accountID, owner)  
    {  
        _creditLimit = creditLimit;  
    }  
    public decimal CurrentBalance { get; private set; } = 0;  
  
    public void Deposit(decimal amount)  
    {  
        if (amount < 0)  
        {  
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit amount must be positive");  
        }  
        CurrentBalance += amount;  
    }  
  
    public void Withdrawal(decimal amount)  
    {  
        if (amount < 0)  
        {  
            throw new ArgumentOutOfRangeException(nameof(amount), "Withdrawal amount must be positive");  
        }  
        if (CurrentBalance - amount < -_creditLimit)  
        {  
            throw new InvalidOperationException("Insufficient funds for withdrawal");  
        }  
        CurrentBalance -= amount;  
    }  
  
    public override string ToString() => $"{base.ToString()}, Balance:
```

```
{CurrentBalance}";  
}
```

Hay una posible preocupación con las jerarquías de clases y los constructores principales: es posible crear varias copias de un parámetro de constructor principal, ya que se usa en clases base y derivadas. En el ejemplo de código siguiente se crean dos copias de cada uno de los campos `owner` y `accountID`:

C#

```
public class SavingsAccount(string accountID, string owner, decimal  
interestRate) : BankAccount(accountID, owner)  
{  
    public SavingsAccount() : this("default", "default", 0.01m) { }  
    public decimal CurrentBalance { get; private set; } = 0;  
  
    public void Deposit(decimal amount)  
    {  
        if (amount < 0)  
        {  
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit  
amount must be positive");  
        }  
        CurrentBalance += amount;  
    }  
  
    public void Withdrawal(decimal amount)  
    {  
        if (amount < 0)  
        {  
            throw new ArgumentOutOfRangeException(nameof(amount),  
"Withdrawal amount must be positive");  
        }  
        if (CurrentBalance - amount < 0)  
        {  
            throw new InvalidOperationException("Insufficient funds for  
withdrawal");  
        }  
        CurrentBalance -= amount;  
    }  
  
    public void ApplyInterest()  
    {  
        CurrentBalance *= 1 + interestRate;  
    }  
  
    public override string ToString() => $"Account ID: {accountID}, Owner:  
{owner}, Balance: {CurrentBalance}";  
}
```

La línea resaltada muestra que el método `ToString` usa los *parámetros de constructor principal* (`owner` y `accountID`) en lugar de las *propiedades de clase base* (`Owner` y `AccountID`). El resultado es que la clase derivada `SavingsAccount` crea almacenamiento para esas copias. La copia de la clase derivada es distinta de la propiedad de la clase base. Si se puede modificar la propiedad de clase base, la instancia de la clase derivada no verá esa modificación. El compilador emite una advertencia para los parámetros del constructor principal que se usan en una clase derivada y se pasan a un constructor de clase base. En este caso, la corrección consiste en usar las propiedades de la clase base.

Resumen

Puede usar los constructores principales como mejor se adapte a su diseño. En el caso de las clases y estructuras, los parámetros de constructor principal son parámetros para un constructor que se debe invocar. Puede usarlos para inicializar propiedades. Puede inicializar campos. Esas propiedades o campos pueden ser inmutables o mutables. Puede usarlos en métodos. Son parámetros y los puede usar de la manera que mejor se adapte a su diseño. Puede obtener más información sobre los constructores principales en el artículo de la [guía de programación de C# sobre constructores de instancia](#) y en la [especificación del constructor principal propuesta](#).

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Tutorial: Exploración de características de C# 11: miembros virtuales estáticos en interfaces

Artículo • 09/05/2023

C# 11 y .NET 7 incluyen *miembros virtuales estáticos en interfaces*. Esta característica permite definir interfaces que incluyen [operadores sobrecargados](#) u otros miembros estáticos. Una vez que haya definido interfaces con miembros estáticos, puede usar esas interfaces como [restricciones](#) para crear tipos genéricos que usen operadores u otros métodos estáticos. Incluso si no crea interfaces con operadores sobrecargados, es probable que se beneficie de esta característica y de las clases matemáticas genéricas habilitadas por la actualización del lenguaje.

En este tutorial, aprenderá a:

- ✓ Definir interfaces con miembros estáticos.
- ✓ Usar interfaces para definir clases que implementan interfaces con operadores definidos.
- ✓ Crear algoritmos genéricos que se basen en métodos de interfaz estática.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET 7, que admite C# 11. El compilador de C# 11 está disponible a partir de la [versión 17.3 de Visual Studio 2022](#) o del [SDK de .NET 7](#).

Métodos de interfaz abstracta estática

Comencemos con un ejemplo. El método siguiente devuelve el punto medio de dos números `double`:

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

La misma lógica funcionaría para cualquier tipo numérico: `int`, `short`, `long`, `float` `decimal` o cualquier tipo que represente un número. Debe tener una manera de

usar los operadores `+` y `/`, y de definir un valor para `2`. Puede usar la interfaz `System.Numerics.INumber<TSelf>` para escribir el método anterior como el siguiente método genérico:

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2); // note:
    the addition of left and right may overflow here; it's just for
    demonstration purposes
```

Cualquier tipo que implemente la interfaz `INumber<TSelf>` debe incluir una definición para `operator +` y `operator /`. El denominador se define mediante `T.CreateChecked(2)` a fin de crear el valor `2` para cualquier tipo numérico, lo que obliga al denominador a ser del mismo tipo que los dos parámetros.

`INumberBase<TSelf>.CreateChecked<TOther>(TOther)` crea una instancia del tipo a partir del valor especificado y genera un elemento `OverflowException` si el valor está fuera del intervalo que se puede representar. (Esta implementación tiene el potencial de desbordamiento si `left` y `right` son valores lo suficientemente grandes. Hay algoritmos alternativos que pueden evitar este posible problema).

Los miembros abstractos estáticos se definen en una interfaz con una sintaxis conocida: se agregan los modificadores `static` y `abstract` a cualquier miembro estático que no proporcione una implementación. En el ejemplo siguiente se define una interfaz `IGetNext<T>` que se puede aplicar a cualquier tipo que invalide `operator ++`:

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

La restricción de que el argumento de tipo, `T`, implementa `IGetNext<T>` garantiza que la firma del operador incluya el tipo contenedor o su argumento de tipo. Muchos operadores exigen que sus parámetros coincidan con el tipo o que sean el parámetro de tipo restringido para implementar el tipo contenedor. Sin esta restricción, el operador `++` no se podría definir en la interfaz `IGetNext<T>`.

Puede crear una estructura que cree una cadena de caracteres "A", donde cada incremento agregue otro carácter a la cadena mediante el código siguiente:

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

Por lo general, puede crear cualquier algoritmo en el que quiera definir `++` para que "genere el siguiente valor de este tipo". El uso de esta interfaz genera código y resultados claros:

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

El ejemplo anterior genera el siguiente resultado:

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAA
AAAAAA
AAAAAA
```

En este pequeño ejemplo se muestra la motivación de esta característica. Puede usar sintaxis natural para operadores, valores constantes y otras operaciones estáticas. Puede explorar estas técnicas al crear varios tipos que dependen de miembros estáticos, incluidos los operadores sobrecargados. Defina las interfaces que coincidan con las funcionalidades de los tipos y, a continuación, declare la compatibilidad de esos tipos con la nueva interfaz.

Matemáticas genéricas

El escenario que inspira el permitir métodos estáticos, incluidos los operadores, en interfaces es la compatibilidad con algoritmos [matemáticos genéricos](#). La biblioteca de clases base de .NET 7.0 contiene definiciones de interfaz para muchos operadores aritméticos, así como interfaces derivadas que combinan muchos operadores aritméticos en una interfaz `INumber<T>`. Vamos a aplicar esos tipos para crear un registro `Point<T>` que pueda usar cualquier tipo numérico para `T`. Algunos objetos `XOffset` y `YOffset` pueden mover el punto con el operador `+`.

Empiece por crear una nueva aplicación de consola, ya sea mediante `dotnet new` o Visual Studio. Establezca el lenguaje C# en "versión preliminar", que habilita las características en vista previa de C# 11. Agregue el siguiente elemento al archivo `csproj` dentro de un elemento `<PropertyGroup>`:

XML

```
<LangVersion>preview</LangVersion>
```

ⓘ Nota

Este elemento no se puede establecer mediante la interfaz de usuario de Visual Studio. Debe editar el archivo del proyecto directamente.

La interfaz pública de `Translation<T>` y `Point<T>` debe parecerse al código siguiente:

C#

```
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

Use el tipo `record` para los tipos `Translation<T>` y `Point<T>`: ambos almacenan dos valores y representan el almacenamiento de datos en lugar de un comportamiento sofisticado. La implementación de `operator +` se parecería al código siguiente:

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

Para que el código anterior se compile, deberá declarar que `T` admite la interfaz `IAdditionOperators<TSelf, TOther, TResult>`. Esta interfaz incluye el método estático `operator +`. Declara tres parámetros de tipo: uno para el operando izquierdo, otro para el operando derecho y otro para el resultado. Algunos tipos implementan `+` para diferentes operandos y tipos de resultado. Agregue una declaración de que el argumento de tipo `T` implementa `IAdditionOperators<T, T, T>`:

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

Después de agregar esa restricción, la clase `Point<T>` puede usar `+` para su operador de suma. Agregue la misma restricción en la declaración `Translation<T>`:

C#

```
public record Translation<T>(T XOffset, T YOffset) where T :
IAdditionOperators<T, T, T>;
```

La restricción `IAdditionOperators<T, T, T>` impide que un desarrollador use la clase para crear un elemento `Translation` mediante un tipo que no cumpla la restricción de la suma a un punto. Ha agregado las restricciones necesarias al parámetro de tipo para `Translation<T>` y `Point<T>`, así que este código funciona. Para probarlo, agregue código similar al siguiente encima de las declaraciones de `Translation` y `Point` en el archivo `Program.cs`:

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

Puede hacer que este código sea más reutilizable declarando que estos tipos implementan las interfaces aritméticas adecuadas. El primer cambio que se va a realizar

es declarar que `Point<T, T>` implementa la interfaz `IAdditionOperators<Point<T>, Translation<T>, Point<T>>`. El tipo `Point` usa diferentes tipos para los operandos y el resultado. El tipo `Point` ya implementa un elemento `operator +` con esa firma, por lo que todo lo que debe hacer es agregar la interfaz a la declaración:

C#

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Por último, al realizar la suma, resulta útil tener una propiedad que defina el valor de identidad aditivo de ese tipo. Hay una nueva interfaz para esa característica:

`IAdditiveIdentity<TSelf, TResult>`. Una traducción de `{0, 0}` es la identidad aditiva: el punto resultante es el mismo que el operando izquierdo. La interfaz `IAdditiveIdentity<TSelf, TResult>` define una propiedad de solo lectura, `AdditiveIdentity`, que devuelve el valor de identidad. El elemento `Translation<T>` necesita algunos cambios para implementar esta interfaz:

C#

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
    IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
            T.AdditiveIdentity);
}
```

Hay algunos cambios aquí, así que vamos a revisarlos uno por uno. En primer lugar, se declara que el tipo `Translation` implementa la interfaz `IAdditiveIdentity`:

C#

```
public record Translation<T>(T XOffset, T YOffset) :
    IAdditiveIdentity<Translation<T>, Translation<T>>
```

A continuación, podría intentar implementar el miembro de interfaz tal y como se muestra en el código siguiente:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

El código anterior no se compilará, ya que `0` depende del tipo. La respuesta: use `IAdditiveIdentity<T>.AdditiveIdentity` para `0`. Este cambio significa que las restricciones deben incluir ahora que `T` implementa `IAdditiveIdentity<T>`. Esto da como resultado la siguiente implementación:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
    T.AdditiveIdentity);
```

Ahora que ha agregado esa restricción en `Translation<T>`, debe agregar la misma restricción a `Point<T>`:

C#

```
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
    Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right)
=>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset
    };
}
```

Este ejemplo le ha permitido ver cómo se componen las interfaces matemáticas genéricas. Ha aprendido a:

- ✓ Escribir un método que depende de la interfaz `INumber<T>`, por lo que se podría usar con cualquier tipo numérico.
- ✓ Crear un tipo que se basa en las interfaces de suma para implementar un tipo que solo admite una operación matemática. Ese tipo declara su compatibilidad con esas mismas interfaces para que se pueda componer de otras maneras. Los algoritmos se escriben con la sintaxis más natural de los operadores matemáticos.

Experimente con estas características y registre comentarios. Puede usar el elemento de menú *Enviar comentarios* de Visual Studio o crear un nuevo [problema](#) en el repositorio roslyn de GitHub. Cree algoritmos genéricos que funcionen con cualquier

tipo numérico. Cree algoritmos con estas interfaces en las que el argumento de tipo solo pueda implementar un subconjunto de funcionalidades de tipo numérico. Aunque no cree interfaces que usen estas capacidades, puede experimentar con ellas en los algoritmos.

Consulte también

- [Matemáticas genéricas](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Uso de la coincidencia de patrones para crear el comportamiento de la clase y mejorar el código

Artículo • 09/05/2023

Las características de coincidencia de patrones en C# proporcionan la sintaxis para expresar los algoritmos. Puede usar estas técnicas para implementar el comportamiento en las clases. Puede combinar el diseño de clases orientadas a objetos con una implementación orientada a datos para proporcionar un código conciso al modelar objetos del mundo real.

En este tutorial, aprenderá a:

- ✓ Expresar las clases orientadas a objetos mediante patrones de datos.
- ✓ Implementar dichos patrones con las características de coincidencia de patrones de C#.
- ✓ Aprovechar los diagnósticos del compilador para validar la implementación.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET 5, incluido el compilador de C# 9. El compilador de C# 9 está disponible a partir de la [versión 16.8 de Visual Studio 2019](#) o del [SDK de .NET 5](#).

Compilación de una simulación de una esclusa de canal

En este tutorial, compilará una clase de C# que simula una [esclusa de canal](#). En pocas palabras, una esclusa de canal es un dispositivo que permite que los barcos suban y bajen cuando se mueven entre dos cuerpos de agua con distintos niveles. Una esclusa tiene dos compuertas y algún mecanismo que permite cambiar el nivel del agua.

En su funcionamiento habitual, un barco entra por una de las compuertas mientras el nivel del agua en la esclusa se iguala al nivel del agua del lado por donde entra el barco. Una vez en la esclusa, se modifica el nivel del agua para igualarlo al nivel de donde el barco saldrá de la esclusa. Cuando el nivel del agua es igual al de ese lado, se abre la compuerta de la salida. Hay medidas de seguridad que garantizan que un operador no pueda crear una situación de peligro en el canal. El nivel del agua solo se puede

modificar cuando ambas compuertas están cerradas. Como mucho, puede haber una compuerta abierta. Para abrir una compuerta, el nivel del agua de la esclusa debe coincidir con el nivel fuera de la compuerta que se está abriendo.

Es posible compilar una clase de C# que modele este comportamiento. Una clase `CanalLock` podría admitir los comandos para abrir o cerrar cualquiera de las compuertas. Tendría otros comandos para aumentar o reducir el nivel del agua. La clase también debería admitir propiedades para leer el estado actual de ambas compuertas y el nivel del agua. Los métodos implementan las medidas de seguridad.

Definición de una clase

Compilará una aplicación de consola para probar la clase `CanalLock`. Cree un proyecto de consola nuevo para .NET 5 con Visual Studio o la CLI de .NET. Luego, agregue una clase nueva con el nombre `CanalLock`. Luego, diseñe la API pública, pero deje los métodos sin implementar:

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
    WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }
}
```

```

    }

    public override string ToString() =>
        $"The lower gate is {((LowWaterGateOpen ? "Open" : "Closed"))}. " +
        $"The upper gate is {((HighWaterGateOpen ? "Open" : "Closed"))}. " +
        $"The water level is {CanalLockWaterLevel}.";
}

```

El código anterior inicializa el objeto de manera que ambas compuertas estén cerradas y el nivel del agua sea bajo. Luego, escriba el código de prueba siguiente en el método `Main` como guía para crear una primera implementación de la clase:

C#

```

// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

```

A continuación, agregue una primera implementación de cada método en la clase `CanalLock`. El código siguiente implementa los métodos de la clase sin preocuparse por las reglas de seguridad. Más adelante agregará pruebas de seguridad:

```
C#  
  
// Change the upper gate.  
public void SetHighGate(bool open)  
{  
    HighWaterGateOpen = open;  
}  
  
// Change the lower gate.  
public void SetLowGate(bool open)  
{  
    LowWaterGateOpen = open;  
}  
  
// Change water level.  
public void SetWaterLevel(WaterLevel newLevel)  
{  
    CanalLockWaterLevel = newLevel;  
}
```

Las pruebas que ha escrito hasta el momento se completan correctamente. Ha implementado los aspectos básicos. Ahora, escriba una prueba para la primera condición de error. Al final de las pruebas anteriores, ambas compuertas están cerradas y el nivel del agua se establece en bajo. Agregue una prueba para intentar abrir la compuerta superior:

```
C#  
  
Console.WriteLine("=====");  
Console.WriteLine("      Test invalid commands");  
// Open "wrong" gate (2 tests)  
try  
{  
    canalGate = new CanalLock();  
    canalGate.SetHighGate(open: true);  
}  
catch (InvalidOperationException)  
{  
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is low.");  
}  
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Esta prueba genera un error porque la compuerta se abre. Como primera implementación, puede corregirlo con este código:

C#

```
// Change the upper gate.  
public void SetHighGate(bool open)  
{  
    if (open && (CanalLockWaterLevel == WaterLevel.High))  
        HighWaterGateOpen = true;  
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))  
        throw new InvalidOperationException("Cannot open high gate when the  
water is low");  
}
```

Las pruebas se realizan correctamente. Pero a medida que agrega más pruebas, agregará cada vez más cláusulas `if` y probará distintas propiedades. Pronto, estos métodos resultarán demasiado complicados a medida que agregue más condicionales.

Implementación de los comandos con patrones

Una mejor manera es usar *patrones* para determinar si el objeto tiene un estado válido para ejecutar un comando. Puede expresar si se permite un comando como una función de tres variables: el estado de la compuerta, el nivel del agua y la nueva configuración:

Nueva configuración	Estado de la compuerta	Nivel del agua	Resultado
Cerrada	Cerrada	Alto	Cerrada
Cerrada	Cerrada	Bajo	Cerrada
Cerrada	Abrir	Alto	Cerrada
Closed	Abrir	Baja	Closed
Abrir	Closed	Alto	Abrir
Abrir	Closed	Bajo	Cerrada (error)
Abrir	Abrir	Alto	Abrir
Abrir	Abrir	Baja	Cerrada (error)

La cuarta y la última fila de la tabla tienen tachado el texto porque no son válidas. El código que va a agregar ahora debe garantizar que la compuerta superior no se abrirá nunca si el nivel del agua es bajo. Esos estados se pueden codificar como una expresión switch única (recuerde que `false` indica "Cerrada"):

C#

```

HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
    InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};

```

Pruebe esta versión. Las pruebas se realizan correctamente, lo que valida el código. En la tabla completa se muestran las combinaciones posibles de entradas y resultados. Eso significa que tanto usted como otros desarrolladores pueden examinar rápidamente la tabla y ver que se han cubierto todas las entradas posibles. Usar el compilador puede hacerlo más fácil. Después de agregar el código anterior, puede ver que el compilador genera una advertencia: CS8524 indica que la expresión `switch` no cubre todas las entradas posibles. El motivo de esta advertencia es que una de las entradas es de tipo `enum`. El compilador interpreta "todas las entradas posibles" como todas las entradas del tipo subyacente, por lo general, un `int`. Esta expresión `switch` solo comprueba los valores declarados en la `enum`. Para quitar la advertencia, puede agregar un patrón de descarte comodín para el último segmento de la expresión. Esta condición genera una excepción porque indica una entrada no válida:

C#

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

El segmento modificador anterior debe ir al final de la expresión `switch` porque coincide con todas las entradas. Experimente poniendo el segmento modificador antes en la expresión. Eso generará un error de compilador CS8510 para un código inalcanzable en un patrón. La estructura natural de las expresiones `switch` permite que el compilador genere errores y advertencias en caso de posibles errores. La "red de seguridad" del compilador facilita la creación de código correcto en menos iteraciones y brinda la libertad de combinar segmentos modificadores con caracteres comodín. El compilador emitirá errores si la combinación da como resultado segmentos inaccesibles no esperados y advertencias si quita un segmento necesario.

El primer cambio consiste en combinar todos los segmentos en los que el comando va a cerrar la compuerta; eso siempre se permite. Agregue el código siguiente como el primer segmento de la expresión `switch`:

C#

```
(false, _, _) => false,
```

Después de agregar el segmento modificador anterior, recibirá cuatro errores de compilador, uno en cada uno de los segmentos donde el comando es `false`. Estos segmentos ya los cubre el segmento recién agregado. Puede quitar sin problemas esas cuatro líneas. Su intención era que este segmento modificador nuevo reemplazara esas condiciones.

Luego, puede simplificar los cuatro segmentos en los que el comando indica abrir la compuerta. En ambos casos en los que el nivel del agua es alto, se puede abrir la compuerta. (En un caso, ya está abierta). Un caso en el que el nivel del agua es bajo genera una excepción y el otro no debería ocurrir. Debería ser seguro generar la misma excepción si el cierre hidráulico ya tiene un estado no válido. Puede hacer estas simplificaciones para esos segmentos:

C#

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Vuelva a ejecutar las pruebas y las completarán correctamente. Esta es la versión final del método `SetHighGate`:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
switch
{
    (false, _, _)          => false,
    (true, _, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
                                     => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

Implementación de patrones por su cuenta

Ahora que conoce la técnica, complete usted mismo los métodos `SetLowGate` y `SetWaterLevel`. Empiece agregando el código siguiente para probar las operaciones no válidas en esos métodos:

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Vuelva a ejecutar la aplicación. Puede ver que se generan errores en las pruebas nuevas y que la esclusa de canal queda en un estado no válido. Intente implementar usted mismo el resto de los métodos. El método para establecer la compuerta inferior debe ser similar al que se usa para establecer la compuerta superior. El método que cambia el nivel del agua tiene otras comprobaciones, pero debe seguir una estructura similar. Puede que le resulte útil usar el mismo proceso para el método que establece el nivel del agua. Comience con las cuatro entradas: El estado de ambas compuertas, el estado actual del nivel del agua y el nuevo nivel del agua solicitado. La expresión switch debe empezar por:

C#

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};
```

Tendrá que completar un total de 16 segmentos modificadores. Luego, realice la prueba y simplifique.

¿Hizo métodos como este?

C#

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
```

```
        InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
    InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    );
}
```

Las pruebas deberían completarse correctamente y la esclusa de canal debería funcionar de manera segura.

Resumen

En este tutorial, aprendió a usar la coincidencia de patrones para comprobar el estado interno de un objeto antes de aplicar cualquier cambio en ese estado. Puede comprobar combinaciones de propiedades. Una vez que cree tablas para cualquiera de esas transiciones, probará el código y, luego, lo simplificará para su lectura y mantenimiento. Estas refactorizaciones iniciales pueden sugerir otras refactorizaciones que validen el estado interno o administren otros cambios de la API. En este tutorial se combinan clases y objetos con un enfoque más orientado a datos basado en patrones para implementar esas clases.

Tutorial: Escritura de un controlador de interpolación de cadenas personalizado

Artículo • 06/04/2023

En este tutorial, aprenderá a:

- ✓ Implementar el patrón del controlador de interpolación de cadenas.
- ✓ Interactuar con el receptor en una operación de interpolación de cadenas.
- ✓ Agregar argumentos al controlador de interpolación de cadenas.
- ✓ Comprender las nuevas características de la biblioteca para la interpolación de cadenas.

Prerrequisitos

Deberá configurar la máquina para ejecutar .NET 6, incluido el compilador de C# 10. El compilador de C# 10 está disponible a partir de [Visual Studio 2022](#) o del [SDK de .NET 6](#).

En este tutorial se da por supuesto que conoce bien C# y .NET, incluidos Visual Studio o la CLI de .NET.

Nuevo esquema

C# 10 agrega compatibilidad con un [*controlador de cadenas interpoladas personalizado*](#). Un controlador de cadenas interpoladas es un tipo que procesa la expresión del marcador de posición en una cadena interpolada. Sin un controlador personalizado, los marcadores de posición se procesan de forma similar a [String.Format](#). Cada marcador de posición tiene formato de texto y, luego, los componentes se concatenan para formar la cadena resultante.

Puede escribir un controlador para cualquier escenario en el que use información sobre la cadena resultante. ¿Se usará? ¿Qué restricciones hay en el formato? Estos son algunos ejemplos:

- Es posible que no necesite que ninguna de las cadenas resultantes sea mayor de algún límite, por ejemplo, 80 caracteres. Puede procesar las cadenas interpoladas para llenar un búfer de longitud fija y detener el procesamiento una vez que se alcanza esa longitud del búfer.
- Puede tener un formato tabular y cada marcador de posición debe tener una longitud fija. Un controlador personalizado puede aplicar esto, en lugar de obligar

a que todo el código de cliente se ajuste.

En este tutorial, creará un controlador de interpolación de cadenas para uno de los escenarios de rendimiento principales: las bibliotecas de registro. Según el nivel de registro configurado, se puede prescindir del trabajo para construir un mensaje de registro. Si el registro está desactivado, no es necesario el trabajo para construir una cadena a partir de una expresión de cadena interpolada. El mensaje nunca se imprime, por lo que se puede omitir cualquier concatenación de cadenas. Además, no es necesario realizar las expresiones usadas en los marcadores de posición, incluida la generación de seguimientos de pila.

Un controlador de cadena interpolada puede determinar si se usará la cadena con formato, y realizar solo el trabajo necesario.

Implementación inicial

Partiremos de una clase `Logger` básica que admite distintos niveles:

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

Este elemento `Logger` admite seis niveles diferentes. Cuando un mensaje no pasa el filtro de nivel de registro, no hay ninguna salida. La API pública del registrador acepta una cadena (con formato completo) como mensaje. Ya se ha realizado todo el trabajo para crear la cadena.

Implementación del patrón del controlador

Este paso consiste en crear un *controlador de cadenas interpoladas* que vuelva a crear el comportamiento actual. Un controlador de cadenas interpoladas es un tipo que debe tener las siguientes características:

- Tener aplicado `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute` al tipo.
- Un constructor que tenga dos parámetros `int: literalLength` y `formatCount`. (Se permiten más parámetros).
- Un método `AppendLiteral` público con la firma `public void AppendLiteral(string s)`.
- Un método `AppendFormatted` público genérico con la firma `public void AppendFormatted<T>(T t)`.

Internamente, el compilador crea la cadena con formato y proporciona un miembro para que un cliente recupere esa cadena. El código siguiente muestra un tipo `LogInterpolatedStringHandler` que satisface estos requisitos:

C#

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int
formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\\tliteral length: {literalLength},
formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\\tAppendLiteral called: {{s}}");

        builder.Append(s);
        Console.WriteLine($"\\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\\tAppendFormatted called: {{t}} is of type
{typeof(T)}");
    }
}
```

```
        builder.Append(t?.ToString());
        Console.WriteLine($"\\tAppended the formatted object");
    }

    internal string GetFormattedText() => builder.ToString();
}
```

Ahora puede agregar una sobrecarga a `LogMessage` en la clase `Logger` para probar el nuevo controlador de cadenas interpoladas:

```
C#

public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

No es necesario quitar el método `LogMessage` original; el compilador prefiere un método con un parámetro de controlador interpolado antes que un método con un parámetro `string` cuando el argumento es una expresión de cadena interpolada.

Puede comprobar que se invoca el nuevo controlador mediante el código siguiente como programa principal:

```
C#

var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This
won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a
string, not an interpolated string expression.");
```

La ejecución de la aplicación genera una salida similar al texto siguiente:

PowerShell

```
literal length: 65, formattedCount: 1
AppendLiteral called: {Error Level. CurrentTime: }
Appended the literal string
AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
```

```
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    Appended the literal string
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This won't be printed.}
    Appended the literal string
Warning Level. This warning is a string, not an interpolated string
expression.
```

Al realizar el seguimiento mediante la salida, puede ver cómo el compilador agrega código para llamar al controlador y compilar la cadena:

- El compilador agrega una llamada para construir el controlador y pasa la longitud total del texto literal en la cadena de formato y el número de marcadores de posición.
- El compilador agrega llamadas a `AppendLiteral` y `AppendFormatted` para cada sección de la cadena literal y para cada marcador de posición.
- El compilador invoca el método `LogMessage` utilizando `CoreInterpolatedStringHandler` como argumento.

Por último, observe que la última advertencia no invoca el controlador de cadenas interpoladas. El argumento es un valor `string`, por lo que la llamada invoca a la otra sobrecarga con un parámetro de cadena.

Adición de más funcionalidades al controlador

La versión anterior del controlador de cadenas interpoladas implementa el patrón. Para evitar el procesamiento de cada expresión de marcador de posición, necesitará más información en el controlador. En esta sección, mejorará el controlador para que haga menos trabajo cuando la cadena construida no se escriba en el registro. Se usará `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute` para especificar una asignación entre parámetros a una API pública y parámetros al constructor de un controlador. De esta forma, el controlador tendrá la información necesaria para determinar si se debe evaluar la cadena interpolada.

Comencemos con los cambios en el controlador. En primer lugar, agregue un campo para comprobar si el controlador está habilitado. Agregue dos parámetros al constructor: uno para especificar el nivel de registro de este mensaje y el otro una referencia al objeto de registro:

C#

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

A continuación, use el campo para que el controlador solo anexe literales u objetos con formato cuando se use la cadena final:

C#

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\\tAppendLiteral called: {{s}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\\tAppendFormatted called: {{t}} is of type
{typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\\tAppended the formatted object");
}
```

A continuación, deberá actualizar la declaración `LogMessage` para que el compilador pase los parámetros adicionales al constructor del controlador. Para ello, se usa `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute` en el argumento del controlador:

C#

```
public void LogMessage(LogLevel level,
[InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
```

```
        Console.WriteLine(builder.GetFormattedText());
    }
```

Este atributo especifica la lista de argumentos para `LogMessage` que se asignan a los parámetros que siguen a los parámetros `literalLength` y `formattedCount` necesarios. La cadena vacía (""), especifica el receptor. El compilador sustituye el valor del objeto `Logger` que representa `this` por el siguiente argumento para el constructor del controlador. El compilador sustituye el valor de `level` por el siguiente argumento. Puede proporcionar cualquier número de argumentos para cualquier controlador que escriba. Los argumentos que agregue son argumentos de cadena.

Puede ejecutar esta versión con el mismo código de prueba. Esta vez, verá los siguientes resultados:

PowerShell

```
literal length: 65, formattedCount: 1
AppendLiteral called: {Error Level. CurrentTime: }
Appended the literal string
AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string
expression.
```

Puede ver que se llama a los métodos `AppendLiteral` y `AppendFormat`, pero no están haciendo nada. El controlador ha determinado que la cadena final no será necesaria, por lo que el controlador no la compila. Todavía hay que realizar un par de mejoras.

En primer lugar, puede agregar una sobrecarga de `AppendFormatted` que restrinja el argumento a un tipo que implementa `System.IFormattable`. Esta sobrecarga permite a los autores de llamada agregar cadenas de formato a los marcadores de posición. Al realizar este cambio, vamos a cambiar también el tipo de valor devuelto de los otros métodos, `AppendFormatted` y `AppendLiteral`, de `void` a `bool` (si alguno de estos métodos tiene tipos de valor devuelto diferentes, se producirá un error de compilación). Ese cambio permite el *cortocircuito*. Los métodos devuelven `false` para indicar que se

debe detener el procesamiento de la expresión de cadena interpolada. Al devolver `true` se indica que se debe continuar. En este ejemplo, se usa para detener el procesamiento cuando la cadena resultante no es necesaria. El cortocircuito admite acciones más específicas. Podría detener el procesamiento de la expresión una vez que alcance una longitud determinada con el fin de admitir búferes de longitud fija. O bien, alguna condición podría indicar que los elementos restantes no son necesarios.

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\\tAppendFormatted (IFormattable version) called: {t}
with format {{format}} is of type {typeof(T)}");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\\tAppended the formatted object");
}
```

Con esa adición, puede especificar cadenas de formato en la expresión de cadena interpolada:

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The
time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This
won't be printed.");
```

La `:t` del primer mensaje especifica el "formato de hora corto" de la hora actual. En el ejemplo anterior se mostraba una de las sobrecargas al método `AppendFormatted` que puede crear para el controlador. No es necesario especificar un argumento genérico para el objeto al que se va a dar formato. Es posible que tenga maneras más eficaces de convertir los tipos creados en cadenas. Puede escribir sobrecargas de `AppendFormatted` que toma esos tipos en lugar de un argumento genérico. El compilador elegirá la mejor sobrecarga. El tiempo de ejecución usa esta técnica para convertir `System.Span<T>` en una salida de cadena. Puede agregar un parámetro entero para especificar la *alineación* de la salida, con o sin `IFormattable`. El elemento `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` que se incluye con .NET 6 contiene nueve sobrecargas de `AppendFormatted` para distintos usos. Puede usarlo como referencia al crear un controlador para sus fines.

Ejecute el ejemplo ahora y verá que, para el mensaje `Trace`, solo se llama al primer `AppendLiteral`:

PowerShell

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use
formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29
PM with format {t} is of type System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string
expression.
```

Puede realizar una actualización final del constructor del controlador que mejore la eficacia. El controlador puede agregar un parámetro `out bool` final. Establecer ese parámetro en `false` indica que no se debe llamar al controlador para procesar la expresión de cadena interpolada:

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

Ese cambio significa que puede quitar el campo `enabled`. Para cambiar el tipo de valor devuelto de `AppendLiteral` y `AppendFormatted` a `void`. Ahora, al ejecutar el ejemplo, verá el resultado siguiente:

PowerShell

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use
formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10
PM with format {t} is of type System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

La única salida cuando se especificaba `LogLevel.Trace` era la salida del constructor. El controlador indicaba que no está habilitado, por lo que no se invocaba ninguno de los métodos `Append`.

En este ejemplo se muestra un punto importante para los controladores de cadena interpolada, especialmente cuando se usan bibliotecas de registro. Es posible que no se produzcan efectos secundarios en los marcadores de posición. Agregue el código siguiente al programa principal y vea este comportamiento en acción:

C#

```
int index = 0;
int numberofIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
{index++}, {index++}, {index++}, {index++}, {index++}");
    numberofIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberofIncrements:
{numberofIncrements}");
```

Puede ver que la variable `index` se incrementa cinco veces con cada iteración del bucle. Dado que los marcadores de posición solo se evalúan para los niveles `Critical`, `Error` y `Warning`, no para `Information` y `Trace`, el valor final de `index` no cumple la expectativa:

PowerShell

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

Los controladores de cadenas interpoladas proporcionan un mayor control sobre cómo se convierte una expresión de cadena interpolada en una cadena. El equipo del entorno de ejecución de .NET ya ha usado esta característica para mejorar el rendimiento en varias áreas. Puede usar la misma funcionalidad en sus propias bibliotecas. Para explorar más a fondo, examine

[System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#), que proporciona una implementación más completa de la que se ha creado aquí. Verá muchas más sobrecargas que son posibles para los métodos `Append`.

Creación de tipos de registros

Artículo • 24/11/2023

Los registros son tipos que usan *igualdad basada en valores*. C# 10 agrega *estructuras de registro* para que pueda definir registros como tipos de valor. Dos variables de un tipo de registro son iguales si las definiciones del tipo de registro son idénticas y si, en cada campo, los valores de ambos registros son iguales. Dos variables de un tipo de clase son iguales si los objetos a los que se hace referencia son el mismo tipo de clase y las variables hacen referencia al mismo objeto. La igualdad basada en valores conlleva otras capacidades que probablemente quiera en los tipos de registros. El compilador genera muchos de esos miembros al declarar un elemento `record` en lugar de `class`. El compilador genera esos mismos métodos para los tipos `record struct`.

En este tutorial, aprenderá a:

- ✓ Decida si agrega el modificador `record` a un tipo `class`.
- ✓ Declarar tipos de registros y tipos de registros posicionales.
- ✓ Reemplazar los métodos por métodos generados por el compilador en los registros.

Prerrequisitos

Tendrá que configurar la máquina para ejecutar .NET 6 o versiones posteriores, incluido el compilador de C# 10 o versiones posteriores. El compilador de C# 10 está disponible a partir de [Visual Studio 2022](#) o del [SDK de .NET 6](#).

Características de los registros

Un *registro* se define al declarar un tipo con la palabra clave `record`, que modifica una declaración `class` o `struct`. Opcionalmente, puede omitir la palabra clave `class` para crear un `record class`. Un registro sigue la semántica de la igualdad basada en valores. Para aplicar la semántica de valores, el compilador genera varios métodos para el tipo de registro (para los tipos `record class` y `record struct`):

- Una invalidación de `Object.Equals(Object)`.
- Un método `Equals` virtual cuyo parámetro es el tipo de registro.
- Una invalidación de `Object.GetHashCode()`.
- Métodos para `operator ==` y `operator !=`.
- Los tipos de registros implementan `System.IEquatable<T>`.

Los registros también proporcionan una invalidación de [Object.ToString\(\)](#). El compilador sintetiza métodos para mostrar registros mediante [Object.ToString\(\)](#). Va a examinar esos miembros a medida que escribe el código de este tutorial. Los registros admiten expresiones `with` para habilitar la mutación no destructiva de registros.

También puede declarar *registros posicionales* mediante una sintaxis más concisa. El compilador sintetiza más métodos automáticamente cuando se declaran registros posicionales:

- Un constructor primario cuyos parámetros coinciden con los parámetros posicionales en la declaración del registro.
- Propiedades públicas para cada parámetro de un constructor primario. Estas propiedades son de *solo inicialización* para los tipos `record class` y `readonly record struct`. Para los tipos `record struct`, son de *lectura y escritura*.
- Un método `Deconstruct` para extraer propiedades del registro.

Compilación de datos de temperatura

Los datos y las estadísticas se encuentran entre los escenarios en los que se recomienda usar registros. En este tutorial va a compilar una aplicación que calcula *grados día* para distintos usos. Los *grados día* son una medida de calor (o falta de él) a lo largo de un período de días, semanas o meses. Los grados día realizan un seguimiento del uso energético y lo predicen. Más días más cálidos significan más aire acondicionado, mientras que más días más fríos implican un mayor uso de calefacción. Los grados día resultan útiles para administrar la población vegetal y poner en correlación su crecimiento a medida que cambiamos de estación. Los grados día ayudan a realizar un seguimiento de las migraciones animales de especies que se desplazan según el clima.

La fórmula se basa en la temperatura media de un día determinado y una temperatura de base de referencia. Para calcular los grados día a lo largo del tiempo, necesita la temperatura mínima y máxima de cada día durante un período de tiempo. Comencemos por crear una nueva aplicación. Cree una nueva aplicación de consola. Cree un nuevo tipo de registro en un nuevo archivo denominado "DailyTemperature.cs":

C#

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp);
```

El código anterior define un *registro posicional*. El registro `DailyTemperature` es un objeto `readonly record struct`, porque el objeto no se hereda de él y debe ser

inmutable. Las propiedades `HighTemp` y `LowTemp` son *propiedades de solo inicialización*, lo que significa que se pueden establecer en el constructor o mediante un inicializador de propiedad. Si quiere que los parámetros posicionales sean de lectura y escritura, declare `record struct` en lugar de `readonly record struct`. El tipo `DailyTemperature` también tiene un *constructor primario* con dos parámetros que coinciden con las dos propiedades. Use el constructor primario para inicializar un registro `DailyTemperature`. En el código siguiente se crea e inicializa varios registros `DailyTemperature`. El primero usa parámetros con nombre para aclarar `HighTemp` y `LowTemp`. Los inicializadores restantes usan parámetros posicionales para inicializar `HighTemp` y `LowTemp`:

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
];
```

Puede agregar sus propias propiedades o métodos a los registros, incluidos los registros posicionales. Tiene que calcular la temperatura media de cada día. Puede agregar esa propiedad al registro `DailyTemperature`:

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Vamos a asegurarnos de que puede usar estos datos. Agregue el código siguiente al método `Main`:

```
C#  
  
foreach (var item in data)  
    Console.WriteLine(item);
```

Ejecute la aplicación y verá un resultado similar a la siguiente pantalla (se han quitado varias filas por motivos de espacio):

```
CLI de .NET  
  
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }  
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }  
  
DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }  
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

El código anterior muestra el resultado de la invalidación de `ToString` sintetizada por el compilador. Si prefiere otro texto, puede escribir una versión propia de `ToString` que impida al compilador sintetizar otra para el usuario.

Cálculo de grados día

Para calcular los grados día, tome la diferencia entre una temperatura de base de referencia y la temperatura media de un día determinado. Para medir el calor a lo largo del tiempo, descarte todos los días en los que la temperatura media esté por debajo de la base de referencia. Para medir el frío a lo largo del tiempo, descarte todos los días en los que la temperatura media esté por encima de la base de referencia. Por ejemplo, en EE. UU. se usa 65F como base de los grados del día de calefacción y refrigeración. Esa es la temperatura a la que no se necesita calefacción ni refrigeración. Si un día tiene una temperatura media de 70F, ese día es 5 grados día de refrigeración y 0 grados día de calefacción. A la inversa, si la temperatura media es de 55F, ese día es 10 grados día de calefacción y 0 grados día de refrigeración.

Estas fórmulas se pueden expresar como una pequeña jerarquía de tipos de registros: un tipo de grado día abstracto y dos tipos concretos de grados días de calefacción y grados día de refrigeración. Estos tipos también pueden ser registros posicionales. Toman una temperatura de base de referencia y una secuencia de registros de temperatura diaria como argumentos del constructor primario:

C#

```
public abstract record DegreeDays(double BaseTemperature,  
IEnumerable<DailyTemperature> TempRecords);  
  
public sealed record HeatingDegreeDays(double BaseTemperature,  
IEnumerable<DailyTemperature> TempRecords)  
    : DegreeDays(BaseTemperature, TempRecords)  
{  
    public double DegreeDays => TempRecords.Where(s => s.Mean <  
BaseTemperature).Sum(s => BaseTemperature - s.Mean);  
}  
  
public sealed record CoolingDegreeDays(double BaseTemperature,  
IEnumerable<DailyTemperature> TempRecords)  
    : DegreeDays(BaseTemperature, TempRecords)  
{  
    public double DegreeDays => TempRecords.Where(s => s.Mean >  
BaseTemperature).Sum(s => s.Mean - BaseTemperature);  
}
```

El registro `DegreeDays` abstracto es la clase base compartida de los registros `HeatingDegreeDays` y `CoolingDegreeDays`. Las declaraciones del constructor primario en los registros derivados muestran cómo administrar la inicialización de registros base. El registro derivado declara parámetros para todos los parámetros del constructor primario del registro base. El registro base declara e inicializa esas propiedades. El registro derivado no las oculta, sino que solo crea e inicializa propiedades para los parámetros que no se han declarado en su registro base. En este ejemplo, los registros derivados no agregan nuevos parámetros del constructor primario. Pruebe el código mediante la adición del código siguiente a su método `Main`:

C#

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);  
Console.WriteLine(heatingDegreeDays);  
  
var coolingDegreeDays = new CoolingDegreeDays(65, data);  
Console.WriteLine(coolingDegreeDays);
```

Se obtiene un resultado similar a la siguiente pantalla:

CLI de .NET

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords =  
record_types.DailyTemperature[], DegreeDays = 85 }  
CoolingDegreeDays { BaseTemperature = 65, TempRecords =  
record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Definición de métodos sintetizados por el compilador

El código calcula el número correcto de grados día de calefacción y refrigeración durante ese período de tiempo. Pero en este ejemplo se muestra por qué es recomendable reemplazar algunos de los métodos sintetizados por registros. Puede declarar su propia versión de cualquiera de los métodos sintetizados por el compilador en un tipo de registro, excepto el método Clone. El método Clone tiene un nombre generado por el compilador y no se puede proporcionar otra implementación. Estos métodos sintetizados incluyen un constructor de copias, los miembros de la interfaz [System.IEquatable<T>](#), las pruebas de igualdad y desigualdad y [GetHashCode\(\)](#). Para los fines de este artículo, se va a sintetizar `PrintMembers`. También puede declarar su propio elemento `ToString`, pero `PrintMembers` constituye una mejor opción para los escenarios de herencia. Para proporcionar su propia versión de un método sintetizado, la firma debe coincidir con el método sintetizado.

El elemento `TempRecords` del resultado de la consola no es útil. Muestra el tipo, pero nada más. Puede cambiar este comportamiento si proporciona su propia implementación del método sintetizado `PrintMembers`. La firma depende de los modificadores aplicados a la declaración de `record`:

- Si un tipo de registro es `sealed`, o `record struct`, la firma es `private bool PrintMembers(StringBuilder builder);`
- Si un tipo de registro no es `sealed` y deriva de `object` (es decir, no declara un registro base), la firma es `protected virtual bool PrintMembers(StringBuilder builder);`
- Si un tipo de registro no es `sealed` y deriva de otro registro, la firma es `protected override bool PrintMembers(StringBuilder builder);`

Estas reglas son más fáciles de asimilar si se entiende el propósito de `PrintMembers`. `PrintMembers` agrega información sobre cada propiedad de un tipo de registro a una cadena. El contrato requiere que los registros base agreguen sus miembros a la pantalla y da por hecho que los miembros derivados van a agregar sus miembros. Cada tipo de registro sintetiza una invalidación de `ToString` que es similar al ejemplo siguiente de `HeatingDegreeDays`:

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
```

```
stringBuilder.Append("HeatingDegreeDays");
stringBuilder.Append(" { ");
if (PrintMembers(stringBuilder))
{
    stringBuilder.Append(" ");
}
stringBuilder.Append("}");
return stringBuilder.ToString();
}
```

Declare un método `PrintMembers` en el registro `DegreeDays` que no imprima el tipo de la colección:

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

La firma declara un método `virtual protected` para que coincida con la versión del compilador. No se preocupe si obtiene los descriptores de acceso incorrectos; el lenguaje aplica la firma correcta. Si olvida los modificadores correctos de cualquier método sintetizado, el compilador emite advertencias o errores que ayudan a obtener la firma correcta.

En C# 10 y versiones posteriores, puede declarar el método `ToString` como `sealed` en un tipo de registro. Esto evita que los registros derivados proporcionen una implementación nueva. Los registros derivados seguirán conteniendo la invalidación de `PrintMembers`. Tendría que sellar `ToString` si no quisiera que mostrara el tipo de entorno de ejecución del registro. En el ejemplo anterior, perdería la información sobre dónde mide el registro los días con temperaturas altas o bajas.

Mutación no destructiva

Los miembros sintetizados de una clase de registro posicional no modifican el estado del registro. El objetivo es que se puedan crear registros inmutables más fácilmente. Recuerde que declara una instancia de `readonly record struct` para crear una estructura de registro inmutable. Vuelva a observar las declaraciones anteriores de `HeatingDegreeDays` y `CoolingDegreeDays`. Los miembros agregados realizan cálculos en los valores del registro, pero no mutan el estado. Los registros posicionales facilitan la creación de tipos de referencia inmutables.

La creación de tipos de referencia inmutables significa que se recomienda usar la mutación no destructiva. Cree nuevas instancias de registro que sean similares a las instancias de registro existentes mediante [with expresiones](#). Estas expresiones son una construcción de copia con asignaciones adicionales que modifican la copia. El resultado es una nueva instancia de registro en la que se ha copiado cada propiedad del registro existente y, opcionalmente, se ha modificado. El registro original no se ha modificado.

Vamos a agregar un par de características al programa que muestren expresiones `with`. En primer lugar, vamos a crear un nuevo registro para calcular la suma térmica con los mismos datos. La *suma térmica* suele usar 41F como base de referencia y mide las temperaturas por encima de ella. Para usar los mismos datos, puede crear un nuevo registro similar a `coolingDegreeDays`, pero con otra temperatura base:

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Puede comparar el número de grados calculado con los números generados con una temperatura de base de referencia superior. Recuerde que los registros son *tipos de referencia* y estas copias son instantáneas. No se copia la matriz de los datos, sino que ambos registros hacen referencia a los mismos datos. Ese hecho supone una ventaja en otro escenario. En el caso de la suma térmica, es útil realizar el seguimiento del total de los cinco días anteriores. Puede crear nuevos registros con otros datos de origen mediante expresiones `with`. En el código siguiente se compila una colección de estas acumulaciones y luego se muestran los valores:

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..
(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

También puede usar expresiones `with` para crear copias de registros. No especifique ninguna propiedad entre las llaves de la expresión `with`. Eso significa crear una copia y no cambiar ninguna propiedad:

C#

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Ejecute la aplicación terminada para ver los resultados.

Resumen

En este tutorial se han mostrado varios aspectos de los registros. Los registros proporcionan una sintaxis concisa para los tipos cuyo uso fundamental es el almacenamiento de datos. En el caso de las clases orientadas a objetos, el uso fundamental es definir responsabilidades. Este tutorial se ha centrado en los *registros posicionales*, donde se puede usar una sintaxis concisa para declarar las propiedades de un registro. El compilador sintetiza varios miembros del registro para copiar y comparar registros. Puede agregar cualquier otro miembro que necesite para sus tipos de registros. Puede crear tipos de registros inmutables sabiendo que ninguno de los miembros generados por el compilador mutaría su estado. Además, las expresiones `with` facilitan la compatibilidad con la mutación no destructiva.

Los registros presentan otra manera de definir tipos. Se usan definiciones `class` para crear jerarquías orientadas a objetos que se centran en las responsabilidades y el comportamiento de los objetos. Cree tipos `struct` para las estructuras de datos que almacenan datos y que son lo suficientemente pequeñas como para copiarse de forma eficaz. Cree tipos `record` si lo que busca son comparaciones y análisis de similitud que se basen en valores, y quiere usar variables de referencia, pero no copiar valores. Los tipos `record struct` se crean cuando se quieren las características de los registros para un tipo lo suficientemente pequeño como para copiarlo de forma eficaz.

Puede obtener más información sobre los registros en el [artículo de referencia del lenguaje C# para el tipo de registro](#) y la [especificación de tipo de registro propuesta](#) y la [especificación de estructura de registro](#).

 Colaborar con nosotros en
GitHub

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Tutorial: Exploración de ideas mediante instrucciones de nivel superior para compilar código mientras aprende

Artículo • 15/11/2023

En este tutorial, aprenderá a:

- ✓ Obtener información sobre las reglas que rigen el uso de las instrucciones de nivel superior.
- ✓ Usar instrucciones de nivel superior para explorar algoritmos.
- ✓ Refactorizar exploraciones en componentes reutilizables.

Requisitos previos

Tendrá que configurar el equipo para ejecutar .NET 6, que incluye el compilador de C# 10. El compilador de C# 10 está disponible a partir de [Visual Studio 2022](#) o del [SDK de .NET 6](#).

En este tutorial se da por supuesto que conoce bien C# y .NET, incluidos Visual Studio o la CLI de .NET.

Comienzo de la exploración

Las instrucciones de nivel superior permiten evitar la ceremonia adicional que requiere colocar el punto de entrada del programa en un método estático en una clase. El punto de partida típico de una aplicación de consola nueva es similar al código siguiente:

```
C#  
  
using System;  
  
namespace Application  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

El código anterior es el resultado de ejecutar el comando `dotnet new console` y crear una aplicación de consola. Estas 11 líneas solo contienen una línea de código ejecutable. Puede simplificar ese programa con la nueva característica de instrucciones de nivel superior. Esto le permite quitar todas las líneas de este programa menos dos:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

ⓘ Importante

Las plantillas de C# para .NET 6 usan *instrucciones de nivel superior*. Es posible que la aplicación no coincida con el código de este artículo si ya ha actualizado a .NET 6. Para obtener más información, consulte el artículo [Las nuevas plantillas de C# generan instrucciones de nivel superior](#).

El SDK de .NET 6 también agrega un conjunto de directivas *implícitas* `global using` para proyectos que usan los SDK siguientes:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Estas directivas de `global using` implícitas incluyen los espacios de nombres más comunes para el tipo de proyecto.

Para saber más, consulte el artículo sobre las [directivas de uso implícito](#)

Esta característica simplifica lo que se necesita para comenzar a explorar nuevas ideas. Puede usar las instrucciones de nivel superior para escenarios de scripting o para explorar. Una vez que conozca los aspectos básicos, puede empezar a refactorizar el código y crear métodos, clases u otros ensamblados para los componentes reutilizables que ha compilado. Las instrucciones de nivel superior permiten una experimentación rápida y tutoriales para principiantes. También proporcionan una ruta fluida desde la experimentación hasta la obtención de programas completos.

Las instrucciones de nivel superior se ejecutan en el orden en el que aparecen en el archivo. Las instrucciones de nivel superior solo se pueden usar en un archivo de código fuente de la aplicación. El compilador genera un error si se usan en más de un archivo.

Creación de un contestador automático de .NET mágico

En este tutorial, se creará una aplicación de consola que responde a una pregunta de tipo "sí" o "no" con una respuesta aleatoria. Compilará la funcionalidad paso a paso. Puede centrarse en la tarea en lugar de la ceremonia necesaria para la estructura de un programa típico. Después, una vez que esté satisfecho con la funcionalidad, puede refactorizar la aplicación como considere oportuno.

Un buen punto de partida es escribir la pregunta de nuevo en la consola. Puede empezar por escribir el código siguiente:

```
C#  
  
Console.WriteLine(args);
```

No declare una variable `args`. Para el único archivo de código fuente que contiene las instrucciones de nivel superior, el compilador reconoce `args` para indicar los argumentos de línea de comandos. El tipo de `args` es `string[]`, como en todos los programas de C#.

Puede ejecutar el comando `dotnet run` siguiente para probar el código:

```
CLI de .NET  
  
dotnet run -- Should I use top level statements in all my programs?
```

Los argumentos después de `--` en la línea de comandos se pasan al programa. Puede ver el tipo de la variable `args`, ya que es lo que se imprime en la consola:

```
Consola  
  
System.String[]
```

Para escribir la pregunta en la consola, tendrá que enumerar los argumentos y separarlos con un espacio. Reemplace la llamada a `WriteLine` por el código siguiente:

```
C#  
  
Console.WriteLine();  
foreach(var s in args)  
{  
    Console.Write(s);
```

```
        Console.Write(' ');
    }
Console.WriteLine();
```

Ahora, al ejecutar el programa, mostrará correctamente la pregunta como una cadena de argumentos.

Respuesta con una respuesta aleatoria

Después de repetir la pregunta, puede agregar el código para generar la respuesta aleatoria. Para empezar, agregue una matriz de respuestas posibles:

```
C#
```

```
string[] answers =
[
    "It is certain.",           "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.",     "Ask again later.",            "My reply is no.",
    "Without a doubt.",        "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",       "Cannot predict now.",         "Outlook not so
good.",
    "You may rely on it.",     "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

Esta matriz tiene diez respuestas que son afirmativas, cinco inexpresivas y cinco negativas. A continuación, agregue el código siguiente para generar y mostrar una respuesta aleatoria de la matriz:

```
C#
```

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Puede volver a ejecutar la aplicación para ver los resultados. Debería ver algo parecido a la salida siguiente:

CLI de .NET

```
dotnet run -- Should I use top level statements in all my programs?
```

Should I use top level statements in all my programs?
Better not tell you now.

Este código responde a las preguntas, pero agreguemos una característica más. Quiere que la aplicación de preguntas simule que se piensa la respuesta. Puede hacerlo si agrega una animación de ASCII y se detiene mientras trabaja. Agregue el código siguiente después de la línea que reproduce la pregunta:

```
C#  
  
for (int i = 0; i < 20; i++)  
{  
    Console.Write("| -");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("/ \\");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("- |");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("\\ /");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
}  
Console.WriteLine();
```

También tendrá que agregar una instrucción `using` a la parte superior del archivo de código fuente:

```
C#  
  
using System.Threading.Tasks;
```

Las instrucciones `using` deben aparecer antes que cualquier otra del archivo. De lo contrario, es un error del compilador. Puede volver a ejecutar el programa y ver la animación. Eso mejora la experiencia. Experimente con la duración del retraso hasta que le guste.

El código anterior crea un conjunto de líneas giratorias separadas por un espacio. Al agregar la palabra clave `await` se le indica al compilador que genere el punto de entrada del programa como un método con el modificador `async` y que devuelva `System.Threading.Tasks.Task`. Este programa no devuelve un valor, por lo que el punto de entrada del programa devuelve `Task`. Si el programa devuelve un valor entero, tendría que agregar una instrucción `return` al final de las instrucciones de nivel superior.

Esa instrucción return especificaría el valor entero que se va a devolver. Si las instrucciones de nivel superior incluyen una expresión `await`, el tipo de valor devuelto se convierte en `System.Threading.Tasks.Task<TResult>`.

Refactorización para el futuro

El programa debería ser similar al código siguiente:

```
C#  
  
Console.WriteLine();  
foreach(var s in args)  
{  
    Console.Write(s);  
    Console.Write(' ');  
}  
Console.WriteLine();  
  
for (int i = 0; i < 20; i++)  
{  
    Console.Write("| -");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("/ \\");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("- |");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("\\ /");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
}  
Console.WriteLine();  
  
string[] answers =  
[  
    "It is certain.",      "Reply hazy, try again.",      "Don't count on  
it.",  
    "It is decidedly so.",  "Ask again later.",          "My reply is no.",  
    "Without a doubt.",    "Better not tell you now.",    "My sources say  
no.",  
    "Yes - definitely.",   "Cannot predict now.",        "Outlook not so  
good.",  
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",  
    "As I see it, yes.",  
    "Most likely.",  
    "Outlook good.",  
    "Yes.",  
    "Signs point to yes.",  
];
```

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

En el código anterior es razonable. Funciona. Pero no es reutilizable. Ahora que ya tiene la aplicación en funcionamiento, es momento de extraer los elementos reutilizables.

Un candidato es el código que muestra la animación en espera. Ese fragmento de código se puede convertir en un método:

Puede empezar por crear una función local en el archivo. Reemplace la animación actual por el código siguiente:

```
C#
```

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

En el código anterior se crea una función local dentro del método Main. Todavía no es reutilizable. Por tanto, extraiga ese código en una clase. Cree un archivo con el nombre *utilities.cs* y agregue el código siguiente:

```
C#
```

```
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
```

```

    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
}
}

```

Un archivo que tiene instrucciones de nivel superior también puede contener espacios de nombres y tipos al final del archivo, después de las instrucciones de nivel superior. Pero para este tutorial, coloque el método de animación en un archivo independiente para que sea más fácil de usar.

Por último, puede limpiar el código de animación para quitar duplicaciones:

```

C#

foreach (string s in animations)
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}

```

Ahora tiene una aplicación completa y ha refactorizado los elementos reutilizables para su uso posterior. Puede llamar al nuevo método de utilidad desde las instrucciones de nivel superior, tal como se muestra a continuación en la versión finalizada del programa principal:

```

C#

using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}

```

```
}

Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",           "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.",     "Ask again later.",            "My reply is no.",
    "Without a doubt.",        "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",       "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",     "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

El ejemplo anterior agrega la llamada a `Utilities.ShowConsoleAnimation` y una instrucción `using` adicional.

Resumen

Las instrucciones de nivel superior facilitan la creación de programas sencillos para explorar nuevos algoritmos. Puede experimentar con algoritmos si prueba otros fragmentos de código. Una vez que haya aprendido lo que funciona, puede refactorizar el código para que sea más fácil de mantener.

Las instrucciones de nivel superior simplifican los programas basados en aplicaciones de consola. Esto incluye Azure Functions, las acciones de GitHub y otras utilidades pequeñas. Para obtener más información, vea [Instrucciones de nivel superior \(Guía de programación de C#\)](#).

 Colaborar con nosotros en
GitHub

El origen de este contenido se
puede encontrar en GitHub,
donde también puede crear y

.NET

Comentarios de .NET

.NET es un proyecto de código
abierto. Seleccione un vínculo para
proporcionar comentarios:

revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

 Abrir incidencia con la documentación

 Proporcionar comentarios sobre el producto

Índices y rangos

Artículo • 14/11/2023

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.
- ✓ Definir implícitamente un objeto [Range](#).
- ✓ 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

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.
- [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.Length - n`.

C#

```
string[] words = [
    // index from start      index from end
    "The",      // 0            ^9
    "quick",    // 1            ^8
    "brown",    // 2            ^7
```

```
"fox",      // 3          ^6
"jumps",    // 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:

C#

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

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.

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.

C#

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

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

C#

```
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:

C#

```
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]:

C#

```
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:

C#

```
int[] numbers = [..Enumerable.Range(0, 100)];
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]},\n
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]}");

```

No solo las matrices admiten índices y rangos. También puede usar índices y rangos con `string`, `Span<T>` o `ReadOnlySpan<T>`.

Conversiones implícitas de expresiones de operador de rango

Cuando se usa la sintaxis de expresión de operador de rango, el compilador convierte implícitamente los valores inicial y final en un objeto `Index` y, a partir de ellos, crea una instancia de `Range`. En el siguiente código se muestra un ejemplo de conversión implícita de la sintaxis de expresión de operador de rango y su alternativa explícita correspondiente:

C#

```

Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//      The implicit range '3..^5' equals the explicit range '3..^5'

```

 **Importante**

Las conversiones implícitas de `Int32` a `Index` producen una excepción `ArgumentOutOfRangeException` cuando el valor es negativo. Del mismo modo, el constructor `Index` produce una excepción `ArgumentOutOfRangeException` cuando el parámetro `value` es negativo.

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 rango 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>`.

ⓘ Importante

El rendimiento del código que usa el operador de rango depende del tipo del operando de la secuencia.

La complejidad temporal del operador de rango depende del tipo de secuencia. Por ejemplo, si la secuencia es un valor `string` o una matriz, el resultado es una copia de la sección especificada de la entrada, por lo que la complejidad temporal es $O(N)$ (donde N es la longitud del rango). Por otro lado, si se trata de `System.Span<T>` o `System.Memory<T>`, el resultado hace referencia a la misma memoria auxiliar, lo que significa que no hay ninguna copia y que la operación es $O(1)$.

Además de la complejidad temporal, esto provoca asignaciones y copias adicionales, lo que afecta al rendimiento. En el código sensible al rendimiento, considere la posibilidad de usar `Span<T>` o `Memory<T>` como el tipo de secuencia, ya que el operador de rango no realiza la asignación.

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, Las matrices multidimensionales no admiten indexadores o rangos. 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:

C#

```
int[][] jagged =
[
    [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],
    [80,81,82,83,84,85,86,87,88,89],
    [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();
}
```

En todos los casos, el operador de rango para [Array](#) asigna una matriz para almacenar los elementos devueltos.

Escenarios para los índices y los rangos

A menudo usará rangos e índices cuando quiera analizar una parte de una secuencia más grande. La nueva sintaxis es más clara al leer exactamente qué parte de la secuencia está implicada. 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:

C#

```
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))];
```

Apunte sobre índices de rango y matrices

Al tomar un rango de una matriz, el resultado es una matriz que se copia de la matriz inicial, en lugar de hacer referencia a esta. Modificar los valores de la matriz resultante no cambiará los de la inicial.

Por ejemplo:

C#

```
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] = 11; // now contains 11,2,3

Console.WriteLine(string.Join(", ", firstThreeItems));
Console.WriteLine(string.Join(", ", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

Consulte también

- [Operadores y expresiones de acceso a miembros](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

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

Artículo • 10/05/2023

Los [tipos de referencia que admiten un valor NULL](#) 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 el equipo para que ejecute .NET, incluido el compilador de C#. El compilador de C# está disponible con [Visual Studio 2022](#) o el [SDK de .NET](#).

En este tutorial se da por supuesto que conoce bien C# y .NET, incluidos Visual Studio o la CLI de .NET.

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`. En proyectos anteriores a C# 11, debe optar por recibir la característica de **tipos de referencia que admiten un valor NULL**. 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:

XML

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

Antes de .NET 6, los nuevos proyectos no incluyen el elemento `Nullable`. A partir de .NET 6, los proyectos nuevos incluyen el elemento `<Nullable>enable</Nullable>` en el archivo del proyecto.

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`:

```
C#
```

```
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:

```
C#
```

```
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:

```
C#
```

```
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:

C#

```
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:

C#

```
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`:

C#

```
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:

C#

```
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:

C#

```
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`:

C#

```
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... AAARGGGGGHHH!";
}
}

```

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`:

C#

```
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`:

C#

```
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`:

C#

```
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`:

C#

```
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`:

C#

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

Obtenga información sobre cómo usar tipos de referencia que aceptan valores NULL al utilizar Entity Framework:

[Aspectos básicos de Entity Framework Core: trabajo con tipos de referencia que aceptan valores NULL](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Uso de la interpolación de cadenas para construir cadenas con formato

Artículo • 10/05/2023

En este tutorial se muestra cómo usar la [interpolación de cadenas](#) de C# para insertar valores en una cadena de resultado único. 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:

CLI de .NET

```
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:

C#

```
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 uno, 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# mostrará un mensaje de error: "error CS1056: Carácter inesperado '\$'").
- Una o varias *expresiones de interpolación*. Una expresión de interpolación se indica mediante una llave de apertura y de cierre ({ }). 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, en primer lugar 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 `publicmodificador 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`:

C#

```
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`:

C#

```
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:

C#

```
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:

C#

```
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 numérica 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:

```
C#  
  
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:

```
C#
```

```
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`.

```
C#
```

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

El resultado tiene un aspecto similar a este:

```
Consola
```

```
[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#

Artículo • 02/09/2023

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](#).

Introducción

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:

```
C#  
  
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);  
// 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:

```
C#  
  
{<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 del resultado de la expresión, siga la expresión de interpolación con dos puntos (":") y la cadena de formato:

```
C#
```

```
{<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:

```
C#
```

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} L. Euler introduced the
letter e to denote {Math.E:F5}.");
// Output:
// On Sunday, November 25, 1731 L. Euler introduced the letter e to denote
2.71828.
```

Para más información, consulte la sección [Componente de cadena de formato](#) del artículo [Formato compuesto](#).

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

Para especificar la anchura mínima del campo y la alineación del resultado de la expresión formateada, siga la expresión de interpolación con una coma (",") y la expresión constante:

```
C#
```

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

Si el valor de *alignment* es positivo, el resultado de la expresión con formato se alinearán 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:

```
C#
```

```
{<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:

```
C#
```

```
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}|");
// Output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetric| 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](#). Una cadena textual interpolada comienza con los \$ caracteres y @. Los tokens \$ y @ se

pueden usar 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 artículo [Formatos compuestos](#).

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

C#

```
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.");
// Output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

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

A partir de C# 11, puede utilizar [literales de cadena crudos interpolados](#).

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:

C#

```
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.

A partir de .NET 6, puede usar el [String.Create\(IFormatProvider, DefaultInterpolatedStringHandler\)](#) método para resolver una cadena interpolada en una cadena de resultado específica de la referencia cultural, como se muestra en el ejemplo siguiente:

C#

```
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;
foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}
{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US      8/27/2023 12:35:31 PM      31,415,926.536
// en-GB      27/08/2023 12:35:31      31,415,926.536
// nl-NL      27-08-2023 12:35:31      31.415.926,536
//          08/27/2023 12:35:31      31,415,926.536
```

En versiones anteriores de .NET, utilice la conversión implícita de una cadena interpolada a una instancia [System.FormattableString](#) y llame a su método [ToString\(IFormatProvider\)](#) para crear una cadena de resultados específica de una cultura. En el siguiente ejemplo se muestra cómo hacerlo:

C#

```
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 = $"{date,23}{number,20:N3}";
    foreach (var culture in cultures)
    {
        var cultureSpecificMessage = message.ToString(culture);
        Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
    }
    // Output is similar to:
    // en-US      8/27/2023 12:35:31 PM      31,415,926.536
    // en-GB      27/08/2023 12:35:31      31,415,926.536
    // nl-NL      27-08-2023 12:35:31      31.415.926,536
    //          08/27/2023 12:35:31      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

A partir de .NET 6, use el [String.Create\(IFormatProvider, DefaultInterpolatedStringHandler\)](#) método para resolver una cadena interpolada en una cadena de resultado para [InvariantCulture](#), como se muestra en el ejemplo siguiente:

C#

```

string message = string.Create(CultureInfo.InvariantCulture, $"Date and time
in invariant culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24

```

En versiones anteriores de .NET, junto con el [FormattableString.ToString\(IFormatProvider\)](#) método , puede usar el método estático [FormattableString.Invariant](#), como se muestra en el ejemplo siguiente:

C#

```

string message = FormattableString.Invariant($"Date and time in invariant
culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// 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](#)

Aplicación de consola

Artículo • 14/03/2023

En este tutorial se enseña una serie de características de .NET y el lenguaje C#.

Aprenderá a realizar los siguientes procedimientos:

- Conceptos básicos de la CLI de .NET
- 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). Puede ejecutar esta aplicación en Windows, Linux, macOS o en un contenedor de Docker.

Hay muchas características en este tutorial. Vamos a compilarlas una a una.

Requisitos previos

- [SDK DE .NET 6](#)
- Un editor de código

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 empezar a realizar modificaciones, vamos a ejecutar la aplicación sencilla Hola mundo. Después de crear la aplicación, escriba `dotnet run` en el símbolo del sistema. Este comando ejecuta el proceso de restauración de paquetes NuGet, crea el archivo ejecutable de la aplicación y lo ejecuta.

Todo el código de la aplicación sencilla Hola mundo está en *Program.cs*. Abra ese archivo con el editor de texto de su elección. Reemplace el código de *Program.cs* por el código siguiente:

C#

```
namespace TeleprompterConsole;

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

En la parte superior del archivo, verá una instrucción `namespace`. 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. Se puede ver que el programa está incluido en el espacio de nombres con el nombre `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. Para obtener información sobre cómo descargar la aplicación de ejemplo de este tutorial, vea las instrucciones en [Ejemplos y tutoriales](#).

Luego, agregue el siguiente método a la clase `Program` (justo debajo del método `Main`):

C#

```
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Este método es un tipo especial de método de C# llamado *método de iterador*. Los métodos de iterador 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 iterador 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 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 llenar el código para leer el archivo en el método `Main`:

```
C#  
  
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 asíncrono. 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:

```
C#  
  
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:

```
C#  
  
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();  
}
```

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:

```
C#  
  
var lineLength = 0;
```

A continuación, agregue el código siguiente después de la instrucción `yield return word + " "`; (antes de la llave de cierre):

```
C#  
  
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`):

```
C#  
  
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 firma 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.

Agregue una palabra clave `await` antes de la llamada a `ShowTeleprompter`:

```
C#
```

```
await ShowTeleprompter();
```

Esto requiere que cambie la firma del método `Main` a:

```
C#
```

```
static async Task Main(string[] args)
```

Obtenga más información sobre el [método `async Main`](#) en nuestra sección de aspectos básicos.

A continuación, debe escribir el segundo método asincrónico para leer desde la Consola y controlar las teclas "<" (menor que), ">" (mayor que), "X" o "x". Este es el método que agrega para esa tarea:

```
C#
```

```
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);
}
```

Esto 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 que se da 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:

C#

```
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`, tal como se ha mostrado anteriormente. También deberá agregar una instrucción `using static` en la parte superior del archivo 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, Esto contrasta

con la instrucción `using` sin `static`, que importa todas las clases de un espacio de nombres.

```
C#
```

```
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:

```
C#
```

```
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:

```
C#
```

```
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 `TeleprompterConfig`. Ahora, debe actualizar `Main` para llamar a `RunTeleprompter` en lugar de a `ShowTeleprompter`:

C#

```
await RunTeleprompter();
```

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.

Para obtener más información sobre la E/S de archivo, vea [E/S de archivos y secuencias](#). Para obtener más información sobre el modelo de programación asincrónica que se ha usado en este tutorial, vea [Programación asincrónica basada en tareas](#) y [Programación asincrónica](#).

Tutorial: Realización de solicitudes HTTP en una aplicación de consola de .NET mediante C#

Artículo • 10/05/2023

En este tutorial se crea una aplicación que emite solicitudes HTTP a un servicio REST en GitHub. La aplicación lee información en formato JSON y convierte la respuesta JSON en objetos de C#. La conversión de JSON en objetos de C# se conoce como *deserialización*.

En el tutorial se muestra cómo hacer lo siguiente:

- ✓ Enviar solicitudes HTTP
- ✓ Deserializar respuestas JSON
- ✓ Configurar la deserialización con atributos

Si prefiere seguir las explicaciones con el [ejemplo final](#) del tutorial, puede descargarlo. Para obtener instrucciones de descarga, vea [Ejemplos y tutoriales](#).

Requisitos previos

- [SDK de .NET 6.0 o versiones posteriores](#).
- Un editor de código como [Visual Studio Code](#) (un editor multiplataforma de código abierto). Puede ejecutar la aplicación de ejemplo en Windows, Linux o macOS, o bien en un contenedor de Docker.

Creación de la aplicación cliente

1. Abra un símbolo del sistema y cree un directorio para la aplicación. Conviértalo en el directorio actual.
2. Escriba el siguiente comando en la ventana de consola:

CLI de .NET

```
dotnet new console --name WebAPIClient
```

Este comando crea los archivos de inicio para una aplicación básica "Hola mundo". El nombre del proyecto es "WebAPIClient".

3. Vaya al directorio "WebAPIClient" y ejecute la aplicación.

```
CLI de .NET
```

```
cd WebAPIClient
```

```
CLI de .NET
```

```
dotnet run
```

`dotnet run` ejecuta automáticamente `dotnet restore` para restaurar las dependencias que necesita la aplicación. También ejecuta `dotnet build` si es necesario. Debería ver la salida de la aplicación "Hello, World!". En el terminal, presione `Ctrl+C` para detener la aplicación.

Realización de solicitudes HTTP

Esta aplicación llama a la [API de GitHub](#) para obtener información sobre los proyectos incluidos bajo el paraguas de [.NET Foundation](#). El extremo es <https://api.github.com/orgs/dotnet/repos>. Para recuperar información, realiza una solicitud HTTP GET. Los exploradores también realizan solicitudes HTTP GET, por lo que puede pegar esa dirección URL en la barra de direcciones del explorador para ver la información que recibirá y procesará.

Use la clase [HttpClient](#) para realizar solicitudes HTTP. [HttpClient](#) solo admite métodos asincrónicos para sus API de larga duración. Por lo tanto, los pasos siguientes crean un método asincrónico y lo llaman desde el método Main.

1. Abra el archivo `Program.cs` en el directorio del proyecto y reemplace su contenido por lo siguiente:

```
C#
```

```
await ProcessRepositoriesAsync();  
  
static async Task ProcessRepositoriesAsync(HttpClient client)  
{  
}
```

Este código:

- Reemplaza la instrucción `Console.WriteLine` por una llamada a `ProcessRepositoriesAsync` que usa la palabra clave `await`.
- Define un método `ProcessRepositoriesAsync` vacío.

2. En la clase `Program`, use un elemento `HttpClient` para controlar solicitudes y respuestas, reemplazando el contenido por el código de C# siguiente.

```
C#  
  
using System.Net.Http.Headers;  
  
using HttpClient client = new();  
client.DefaultRequestHeaders.Accept.Clear();  
client.DefaultRequestHeaders.Accept.Add(  
    new  
    MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));  
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation  
Repository Reporter");  
  
await ProcessRepositoriesAsync(client);  
  
static async Task ProcessRepositoriesAsync(HttpClient client)  
{  
}
```

Este código:

- Configura encabezados HTTP para todas las solicitudes:
 - Un encabezado `Accept` para aceptar respuestas JSON.
 - Encabezado de `User-Agent`. Estos encabezados se comprueban mediante el código de servidor de GitHub y son necesarios para recuperar información de GitHub.

3. En el método `ProcessRepositoriesAsync`, llame al punto de conexión de GitHub que devuelve una lista de todos los repositorios de la organización de .NET Foundation:

```
C#  
  
static async Task ProcessRepositoriesAsync(HttpClient client)  
{  
    var json = await client.GetStringAsync(  
        "https://api.github.com/orgs/dotnet/repos");  
  
    Console.WriteLine(json);  
}
```

Este código:

- Espera la tarea devuelta al llamar al método `HttpClient.GetStringAsync(String)`. Este método envía una solicitud HTTP GET

al URI especificado. El cuerpo de la respuesta se devuelve como un elemento [String](#), que está disponible cuando se completa la tarea.

- La cadena de respuesta `json` se imprime en la consola.

4. Compile la aplicación y ejecútela.

CLI de .NET

```
dotnet run
```

No hay ninguna advertencia de compilación porque `ProcessRepositoriesAsync` ahora contiene un operador `await`. La salida es una larga presentación de texto JSON.

Deserialización del resultado JSON

Los pasos siguientes convierten la respuesta JSON en objetos de C#. La clase [System.Text.Json.JsonSerializer](#) se usa para deserializar JSON en objetos.

1. Cree un archivo con el nombre *Repository.cs* y agregue el código siguiente:

C#

```
public record class Repository(string name);
```

El código anterior define una clase para representar el objeto JSON devuelto desde la API de GitHub. Usará esta clase para mostrar una lista de nombres de repositorio.

El objeto JSON de un objeto de repositorio contiene docenas de propiedades, pero solo se deserializará la propiedad `name`. El serializador omite automáticamente las propiedades JSON para las que no hay ninguna coincidencia en la clase de destino. Esta característica facilita la creación de tipos que funcionan con solo un subconjunto de campos de un paquete JSON grande.

La convención de C# es [poner en mayúscula la primera letra de los nombres de propiedad](#), pero la propiedad `name` comienza aquí con minúscula porque coincide exactamente con lo que hay en JSON. Más adelante verá cómo usar nombres de propiedad de C# que no coinciden con los nombres de propiedad JSON.

2. Use el serializador para convertir JSON en objetos de C#. Reemplace la llamada a `GetStringAsync(String)` en el método `ProcessRepositoriesAsync` por las líneas

siguientes:

```
C#  
  
await using Stream stream =  
    await  
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");  
var repositories =  
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
```

El código actualizado reemplaza `GetStringAsync(String)` por `GetStreamAsync(String)`. Este método del serializador usa como origen una secuencia, en lugar de una cadena.

El primer argumento para `JsonSerializer.DeserializeAsync< TValue >(Stream, JsonSerializerOptions, CancellationToken)` es una expresión `await`. Las expresiones `await` pueden aparecer prácticamente en cualquier parte del código, aunque hasta ahora solo las ha visto como parte de una instrucción de asignación. Los otros dos parámetros, `JsonSerializerOptions` y `CancellationToken`, son opcionales y se omiten en el fragmento de código.

El método `DeserializeAsync` es *genérico*, lo que significa es necesario 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 en `List<Repository>`, que es otro objeto genérico, `System.Collections.Generic.List<T>`. La clase `List<T>` administra una colección de objetos. El argumento de tipo declara el tipo de objetos almacenados en `List<T>`. El argumento de tipo es el registro `Repository`, porque el texto JSON representa una colección de objetos de repositorio.

3. Agregue código para mostrar el nombre de cada repositorio. Reemplace las líneas donde pone:

```
C#  
  
Console.WriteLine(json);
```

por el siguiente:

```
C#  
  
foreach (var repo in repositories ?? Enumerable.Empty<Repository>())  
    Console.WriteLine(repo.name);
```

4. Las directivas `using` siguientes deben situarse en la parte superior del archivo:

```
C#  
  
using System.Net.Http.Headers;  
using System.Text.Json;
```

5. Ejecutar la aplicación.

```
CLI de .NET  
  
dotnet run
```

La salida es una lista con los nombres de los repositorios que forman parte de .NET Foundation.

Configuración de la deserialización

1. En `Repository.cs`, reemplace el contenido del archivo por el código de C# siguiente.

```
C#  
  
using System.Text.Json.Serialization;  
  
public record class Repository(  
    [property: JsonPropertyName("name")] string Name);
```

Este código:

- Cambia el nombre de la propiedad `name` a `Name`.
- Agrega `JsonPropertyNameAttribute` para especificar cómo aparece esta propiedad en el JSON.

2. En `Program.cs`, actualice el código para aplicar el nuevo uso de las mayúsculas de la propiedad `Name`:

```
C#  
  
foreach (var repo in repositories)  
    Console.WriteLine(repo.Name);
```

3. Ejecutar la aplicación.

La salida es la misma.

Refactorizar el código

El método `ProcessRepositoriesAsync` puede realizar el trabajo asíncrono y devolver una colección de los repositorios. Cambie ese método para devolver `Task<List<Repository>>` y mueva el código que escribe en la consola cerca del autor de la llamada.

1. Cambie la firma de `ProcessRepositoriesAsync` para devolver una tarea cuyo resultado sea una lista de objetos `Repository`:

C#

```
static async Task<List<Repository>> ProcessRepositoriesAsync()
```

2. Devuelva los repositorios después de procesar la respuesta JSON:

C#

```
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
return repositories ?? new();
```

El compilador genera el objeto `Task<T>` para el valor devuelto porque ha marcado este método como `async`.

3. Modifique el archivo `Program.cs`, reemplazando la llamada a `ProcessRepositoriesAsync` con lo siguiente para capturar los resultados y escribir cada nombre de repositorio en la consola.

C#

```
var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
    Console.WriteLine(repo.Name);
```

4. Ejecutar la aplicación.

La salida es la misma.

Deserialización de más propiedades

En los pasos siguientes se agrega código para procesar más propiedades del paquete JSON recibido. Probablemente no le interesaría procesar todas las propiedades, pero si agrega algunas más verá una demostración de otras características de C#.

1. Reemplace el contenido de la clase `Repository` por la definición `record` siguiente:

```
C#  
  
using System.Text.Json.Serialization;  
  
public record class Repository(  
    [property: JsonPropertyName("name")] string Name,  
    [property: JsonPropertyName("description")] string Description,  
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,  
    [property: JsonPropertyName("homepage")] Uri Homepage,  
    [property: JsonPropertyName("watchers")] int Watchers);
```

Los tipos `Uri` y `int` tienen una funcionalidad integrada para convertir a y desde la representación de cadena. No se necesita código adicional para deserializar desde el formato de cadena de JSON a esos tipos de destino. Si el paquete JSON contiene datos que no se convierten en un tipo de destino, la acción de serialización genera una excepción.

2. Actualice el bucle `foreach` en el archivo `Program.cs` para mostrar los valores de propiedad:

```
C#  
  
foreach (var repo in repositories)  
{  
    Console.WriteLine($"Name: {repo.Name}");  
    Console.WriteLine($"Homepage: {repo.Homepage}");  
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");  
    Console.WriteLine($"Description: {repo.Description}");  
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");  
    Console.WriteLine();  
}
```

3. Ejecutar la aplicación.

La lista ahora incluye las propiedades adicionales.

Incorporación de una propiedad de fecha

La fecha de la última operación de inserción presenta este formato en la respuesta JSON:

JSON

```
2016-02-08T21:27:00Z
```

Este es el formato de la hora universal coordinada (UTC), por lo que el resultado de la deserialización es un valor [DateTime](#) cuya propiedad [Kind](#) es [Utc](#).

Para que una fecha y hora se represente en su zona horaria, debe escribir un método de conversión personalizado.

1. En *Repository.cs*, agregue una propiedad para la representación UTC de la fecha y hora y una propiedad [LastPush](#) de solo lectura que devuelva la fecha convertida a la hora local. El archivo debería tener el siguiente aspecto:

C#

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name,
    [property: JsonPropertyName("description")] string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
    [property: JsonPropertyName("homepage")] Uri Homepage,
    [property: JsonPropertyName("watchers")] int Watchers,
    [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc)
{
    public DateTime LastPush => LastPushUtc.ToLocalTime();
}
```

La propiedad [LastPush](#) se define utilizando un *miembro con forma de expresión* para el descriptor de acceso [get](#). No hay ningún descriptor de acceso [set](#). La omisión del descriptor de acceso [set](#) es una forma de definir una propiedad *read-only* en C#. (Sí, puede crear propiedades de *solo escritura* en C#, pero su valor es limitado).

2. Agregue de nuevo otra instrucción de salida en *Program.cs*:

C#

```
Console.WriteLine($"Last push: {repo.LastPush}");
```

3. La aplicación completa debe ser similar al archivo *Program.cs* siguiente:

C#

```
using System.Net.Http.Headers;
using System.Text.Json;

using HttpClient client = new();
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 repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
client)
{
    await using Stream stream =
        await
    client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
    (stream);
    return repositories ?? new();
}
```

4. Ejecutar la aplicación.

La salida incluye la fecha y hora de la última inserción en cada repositorio.

Pasos siguientes

En este tutorial, ha creado una aplicación que realiza solicitudes web y analiza los resultados. La versión de la aplicación debe coincidir ahora con el [ejemplo terminado](#).

Obtenga más información sobre cómo configurar la serialización JSON en [Procedimiento para serializar y deserializar \(calcular referencias y resolver referencias\) JSON en .NET](#).

Uso de Language-Integrated Query (LINQ)

Artículo • 10/05/2023

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 mundo".

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`:

```
C#  
  
// 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 llenar 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:

```
C#  
  
// 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:

C#

```

// 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:

C#

```
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.

```
C:\console-1inq>dotnet run
{ Suit = clubs, Rank = two }
{ Suit = clubs, Rank = three }
{ Suit = clubs, Rank = four }
{ Suit = clubs, Rank = five }
{ Suit = clubs, Rank = six }
{ Suit = clubs, Rank = seven }
{ Suit = clubs, Rank = eight }
{ Suit = clubs, Rank = nine }
{ Suit = clubs, Rank = ten }
{ Suit = clubs, Rank = jack }
{ Suit = clubs, Rank = queen }
{ Suit = clubs, Rank = king }
{ Suit = clubs, Rank = ace }
{ Suit = diamonds, Rank = two }
{ Suit = diamonds, Rank = three }
{ Suit = diamonds, Rank = four }
{ Suit = diamonds, Rank = five }
{ Suit = diamonds, Rank = six }
{ Suit = diamonds, Rank = seven }
{ Suit = diamonds, Rank = eight }
{ Suit = diamonds, Rank = nine }
{ Suit = diamonds, Rank = ten }
```

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`:

```
C#  
  
// 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:

```
C#  
  
// 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:

```
C#  
  
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:

```
C#  
  
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, vuelve al método `Main` y ordene la baraja aleatoriamente una vez:

```
C#  
  
// 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:

C#

```

public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!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:

C#

```
// 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:

C#

```
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.

C#

```
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`:

C#

```
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:

C#

```
// 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:

C#

```

public static void Main(string[] args)
{
    IEnumerable<Suit>? suits = Suits();
    IEnumerable<Rank>? ranks = Ranks();

    if ((suits is null) || (ranks is null))
        return;

    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 `ToArrayList` 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

A parte 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](#)

Language-Integrated Query (LINQ)

Artículo • 11/04/2024

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, tendría 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.

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`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<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 usan muchas construcciones familiares del lenguaje C#.
- 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](#).

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto](#) ↗

Introducción a las consultas LINQ en C#

Artículo • 26/04/2024

Una *consulta* es una expresión que recupera datos de un origen de datos. Los distintos orígenes de datos tienen diferentes lenguajes de consulta nativos, por ejemplo SQL para bases de datos relacionales y XQuery para XML. Los programadores deben 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 de lenguaje C# coherente para tipos de orígenes de datos y formatos. En una consulta LINQ, siempre se trabaja con objetos de C#. Use los mismos patrones de codificación básicos para consultar y transformar datos en documentos XML, bases de datos SQL, colecciones de .NET y cualquier otro formato cuando un proveedor LINQ esté disponible.

Las tres partes de una operación de consulta

Todas las operaciones de consulta LINQ constan de tres acciones distintas:

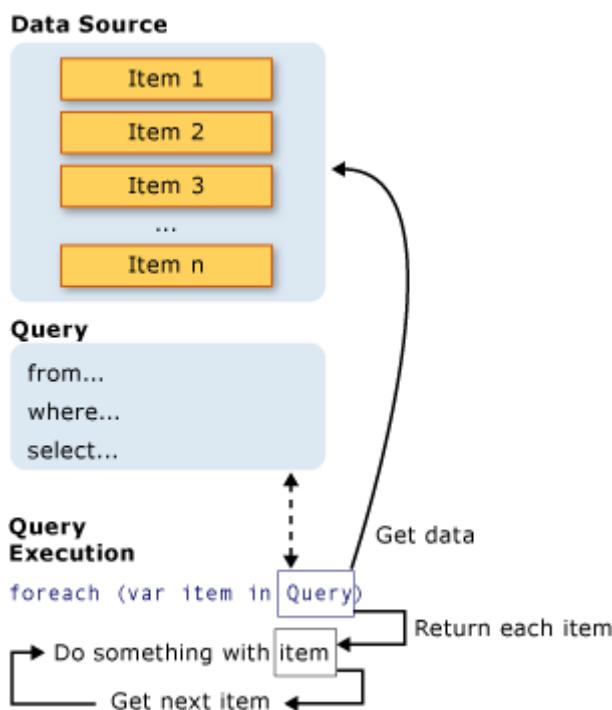
1. Obtener el origen de datos.
2. Crear la consulta.
3. Ejecutar la consulta.

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 artículo.

C#

```
// The Three Parts of a LINQ Query:  
// 1. Data source.  
int[] numbers = [ 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}, ", 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 se recupera ningún dato mediante la creación de una variable de consulta.



El origen de datos

El origen de datos del ejemplo anterior es una matriz, que admite 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 consultable **XElement**:

C#

```
// Create a data source from an XML document.  
// using System.Xml.Linq;  
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Con **EntityFramework**, se crea una asignación relacional de objetos entre las clases de C# y el esquema de la base de datos. Después, se escriben las consultas en los objetos y, en tiempo de ejecución, EntityFramework 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>](#).

C#

```
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 sencilla: un origen de datos de LINQ es cualquier objeto que admita la interfaz genérica [IEnumerable<T>](#) o una interfaz que la haya heredado, normalmente [IQueryable<T>](#).

① Nota

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 \(C#\)](#).

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 dar forma a esa información antes de devolverse. Las consultas se almacenan en una variable de consulta y se inicializan con una expresión de consulta. Use [sintaxis de consulta de C#](#) para escribir consultas.

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, ha observado que la ordenación de las cláusulas se invierte del orden en 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. Todas las cláusulas de consulta se describen en detalle en esta sección. 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, consulte [Información general sobre operadores de consulta estándar \(C#\)](#).

ⓘ Nota

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).

Clasificación de operadores de consulta estándar por modo de ejecución

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 se pueden dividir además en dos categorías: *de streaming* y *de no streaming*.

Inmediato

La ejecución inmediata significa que se lee el origen de datos y que la operación se realiza una vez. Todos los operadores de consulta estándar que devuelven un resultado escalar se ejecutan de manera inmediata. Ejemplos de estas consultas son `Count`, `Max`, `Average` y `First`. Estos métodos se ejecutan sin una instrucción `foreach` explícita porque la propia consulta debe usar `foreach` para devolver un resultado. Estas consultas devuelven un único valor, y no una colección `IEnumerable`. Puede forzar que *cualquier* consulta se ejecute inmediatamente si usa los métodos `Enumerable.ToList` o `Enumerable.ToArray`. La ejecución inmediata permite reutilizar los resultados de la consulta, no su declaración. Los resultados se recuperan una vez y, después, se almacenan para usarlos en el futuro. La consulta siguiente devuelve un recuento de los números pares de la matriz de origen:

C#

```
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`.

```
C#  
  
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.

Aplazado

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`. 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. La ejecución diferida aporta la facilidad de reutilizar las consultas, ya que la consulta captura los datos actualizados del origen de datos cada vez que se iteran los resultados de la consulta. En el código siguiente se muestra un ejemplo de ejecución aplazada:

```
C#  
  
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 repetidamente para recuperar los datos actualizados. Por ejemplo, una aplicación independiente podría actualizar una base de datos continuamente. En *la* aplicación, podría crear una consulta que recupere los datos más recientes y podría ejecutarla a intervalos para recuperar los resultados actualizados.

Los operadores de consulta que usan la ejecución aplazada se pueden clasificar además como de streaming o de no streaming.

Streaming

Los operadores de streaming no deben leer todos los datos de origen antes de que generen 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.

De no streaming

Los operadores de no streaming deben leer todos los datos de origen antes de poder generar 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 generan los elementos resultantes.

Tabla de clasificación

En la tabla siguiente se clasifica cada método de operador de consulta estándar según su método de ejecución.

ⓘ Nota

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.

 Expandir tabla

Operador de consulta estándar	Tipo de valor devuelto	Ejecución inmediata	Ejecución aplazada de streaming	Ejecución aplazada de no streaming
Aggregate	<code>TSource</code>	X		
All	<code>Boolean</code>	X		
Any	<code>Boolean</code>	X		
AsEnumerable	<code>IEnumerable<T></code>		X	
Average	Valor numérico único	x		
Cast	<code>IEnumerable<T></code>		X	
Concat	<code>IEnumerable<T></code>		X	
Contains	<code>Boolean</code>	X		
Count	<code>Int32</code>	X		
DefaultIfEmpty	<code>IEnumerable<T></code>		X	
Distinct	<code>IEnumerable<T></code>		X	
ElementAt	<code>TSource</code>	X		
ElementAtOrDefault	<code>TSource?</code>	X		
Empty	<code>IEnumerable<T></code>	X		
Except	<code>IEnumerable<T></code>		X	X
First	<code>TSource</code>	X		
FirstOrDefault	<code>TSource?</code>	X		
GroupBy	<code>IEnumerable<T></code>			X
GroupJoin	<code>IEnumerable<T></code>		X	X
Intersect	<code>IEnumerable<T></code>		X	X
Join	<code>IEnumerable<T></code>		X	X

Operador de consulta estándar	Tipo de valor devuelto	Ejecución inmediata	Ejecución aplazada de streaming	Ejecución aplazada de no streaming
Last	TSouce	X		
LastOrDefault	TSouce?	X		
LongCount	Int64	X		
Max	Valor numérico único, TSouce o TResult?	x		
Min	Valor numérico único, TSouce 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	TSouce	X		
SingleOrDefault	TSouce?	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
ThenByDescending	IOrderedEnumerable<TElement>			X

Operador de consulta estándar	Tipo de valor devuelto	Ejecución inmediata	Ejecución aplazada de streaming	Ejecución aplazada de no streaming
ToArray	Matriz <code>TSource[]</code>	X		
ToDictionary	<code>Dictionary<TKey,TValue></code>	X		
ToList	<code>IList<T></code>	X		
ToLookup	<code>ILookup<TKey,TElement></code>	X		
Union	<code>IEnumerable<T></code>		X	
Where	<code>IEnumerable<T></code>		X	

LINQ to Objects

"LINQ to Objects" hace referencia al uso de consultas LINQ con cualquier colección `IEnumerable` o `IEnumerable<T>` directamente. 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. En el enfoque de LINQ, se escribe código declarativo que describe qué se quiere recuperar. LINQ to Objects proporciona una excelente introducción a la programación con LINQ.

Las consultas LINQ ofrecen tres ventajas principales con 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.

Cuanto más compleja sea la operación que se quiere realizar en los datos, más ventajas se obtienen al usar LINQ en lugar de las técnicas de iteración tradicionales.

Almacenar los resultados de una consulta en memoria

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](#)

Debe asignar el objeto de colección devuelto a una nueva variable al almacenar los resultados de consulta, como se muestra en el ejemplo siguiente:

C#

```
List<int> numbers = [1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20];

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.
var factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
// data.
Console.WriteLine(factorsofFourList[2]);
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

Consulte también

- [Tutorial: Escribir consultas en C#](#)
- [foreach, in](#)
- [Palabras clave para consultas \(LINQ\)](#)



Colaborar con nosotros en
GitHub



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Conceptos básicos de las expresiones de consultas

Artículo • 06/03/2024

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 los elementos XML 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[]`):

```
C#  
  
IQueryable<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`.

C#

```
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`:

C#

```
var 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 `Enumerable.Count`. También puede usar una nueva variable para almacenar el resultado concreto.

C#

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
var 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 otras cláusulas `from`. También puede usar la palabra clave `into` para que el resultado de una cláusula `join` o `group` actúe como el origen de más cláusulas de consulta 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 genera una secuencia de elementos cuando se itere en una instrucción `foreach` o en una llamada directa a su método `IEnumerator.MoveNext()`.

ⓘ Nota

En los ejemplos de este artículo se usan los siguientes datos de ejemplo y origen de datos.

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
    Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

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.

C#

```
// Data source.
int[] scores = [90, 71, 82, 93, 75, 82];

// Query Expression.
IQueryable<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 (var testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 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:

```

C#

city[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000)
];

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Execute the query to produce the results
foreach (City city in queryMajorCities)
{
    Console.WriteLine(city);
}

// Output:
// City { Population = 120000 }
// City { Population = 112000 }
// City { Population = 150340 }

// 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:

C#

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IQueryable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
var largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IQueryable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
var largeCitiesList2 = largeCitiesQuery.ToList();
```

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

Por ejemplo, la consulta de ejemplo que se mostró anteriormente en este artículo también se puede expresar mediante la asignación implícita de tipos:

```
C#
```

```
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

En el ejemplo anterior, el uso de var es opcional. `queryCities` es un `IEnumerable<City>` que indica si se escribe implícita o explícitamente.

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.

```
C#
```

```
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:

```
C#
```

```
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 tipo `char` con un valor que es la primera letra del nombre de un país.

C#

```
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.

C#

```
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. Los nuevos objetos se inicializan mediante un inicializador de objeto.

C#

```
var queryNameAndPop =
    from country in countries
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };
```

Por lo tanto, en este ejemplo, se requiere el `var` porque la consulta genera un tipo anónimo.

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. Use la cláusula `into` 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, más cláusulas filtran algunos grupos y, después, ordenan los grupos en orden ascendente. Para realizar esas operaciones adicionales, es necesaria la continuación representada por `countryGroup`.

C#

```
// 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 optionales 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.

C#

```
IEnumerable<City> queryCityPop =  
    from city in cities  
    where city.Population is < 200000 and > 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`.

C#

```
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`.

C#

```
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`.

C#

```
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 (var s in queryFirstNames)  
{  
    Console.WriteLine(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.

C#

```
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()  
    };
```

Para más información, consulte [Realizar una subconsulta en una operación de agrupación](#).

Consulte también

- [Palabras clave de consultas \(LINQ\)](#)

- Información general sobre operadores de consulta estándar

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Escritura de consultas LINQ de C# para consultar datos

Artículo • 02/05/2024

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 la sintaxis de consulta suele ser 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. Debería familiarizarse con cómo usar la sintaxis del método en 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*.

C#

```
int[] numbers = [ 5, 10, 8, 3, 6, 12 ];

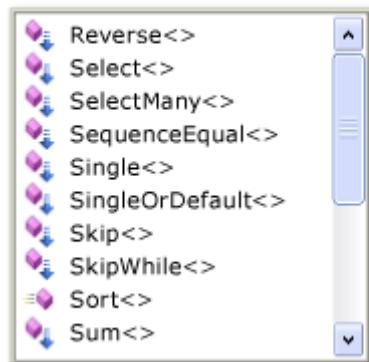
//Query syntax:
IQueryable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;
```

```
//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 + " ");  
}
```

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 tiene un tipo de `IEnumerable<int>`. Si está familiarizado con la interfaz genérica `IEnumerable<T>`, sabrá 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 métodos implementan los operadores de consulta estándar.

```
List<string> list = new List<string>();  
list.|
```



Aunque parece que `IEnumerable<T>` incluye más métodos, no es así. Los operadores de consulta estándar se implementan como *métodos 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 usar métodos de extensión, es necesario incluirlos en el ámbito de las 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 LINQ, como [Entity Framework](#) y LINQ to XML, implementan sus propios operadores de consulta estándar y métodos de extensión para otros tipos 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 `Enumerable.Where`: `Where(num => num % 2 == 0)`. Esta expresión insertada se denomina [expresión lambda](#). Es una manera cómoda de escribir código que, de lo contrario, tendría que escribirse de forma más complicada. 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. Determinadas consultas solo se pueden expresar en sintaxis de método y algunas requieren expresiones lambda. Las expresiones lambda son una herramienta eficaz y flexible en el cuadro de herramientas de LINQ.

Capacidad de composición de consultas

En el ejemplo de código anterior, el método `Enumerable.OrderBy` se invoca mediante el operador de punto en la llamada a `Where`. `Where` genera una secuencia filtrada y, a continuación, `Orderby` ordena la secuencia generada por `Where`. Dado que las consultas devuelven un `IEnumerable`, redáctelas con la sintaxis de método encadenando las llamadas de método. El compilador realiza esta composición al escribir consultas mediante la sintaxis de consulta. Dado que una variable de consulta no almacena los resultados de la consulta, es posible modificarla o usarla como base para una nueva consulta en cualquier momento, incluso después de ejecutarla.

Los ejemplos siguientes muestran algunas consultas LINQ sencillas mediante cada enfoque enumerado anteriormente.

ⓘ Nota

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 mayoría de las consultas se escriben con *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.

C#

```
List<int> numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num is < 3 or > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num is < 3 or > 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];
```

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.

Ejemplo: Sintaxis de método

Algunas operaciones de consulta deben expresarse como una llamada a método. Los más comunes de dichos métodos son aquellos 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 devuelven 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:

C#

```
List<int> numbers1 = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];
List<int> numbers2 = [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 `System.Action` o `System.Func<TResult>`, se proporcionan estos argumentos en forma de `expresión lambda`, tal y como se muestra en el ejemplo siguiente:

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

En las consultas anteriores, solo la consulta n.º 4 se ejecuta inmediatamente, ya que devuelve un solo valor y no una colección `IEnumerable<T>` genérica. El propio método usa `foreach` o código similar para procesar su valor.

Cada una de las consultas anteriores puede escribirse mediante tipado implícito con `var`, tal como se muestra en el ejemplo siguiente:

C#

```
// var is used for convenience in these queries
double 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. Sin embargo, en general es mejor usar una segunda variable para almacenar el resultado de la llamada de método. De esta manera es menos probable que la consulta se confunda con los resultados de la consulta.

C#

```
// Query #7.

// Using a query expression with method syntax
var numCount1 =
    from num in numbers1
    where num is > 3 and < 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num is > 3 and < 7
    select num;

var 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:

C#

```
var numCount = (from num in numbers...
```

Puede escribirse en sintaxis de método como sigue:

```
C#
```

```
var numCount = numbers.Count(n => n is > 3 and < 7);
```

Puede escribirse mediante tipos explícitos como sigue:

```
C#
```

```
int numCount = numbers.Count(n => n is > 3 and < 7);
```

Especificación de filtros con predicado de forma dinámica en tiempo de ejecución

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. La consulta devuelve resultados distintos en función del valor de `id` al ejecutarse la consulta.

```
C#
```

```
int[] ids = [111, 114, 112];

var queryNames =
    from student in students
    where ids.Contains(student.ID)
    select new
    {
        student.LastName,
        student.ID
    };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
   Garcia: 114
   O'Donnell: 112
   Omelchenko: 111
 */

// Change the ids.
ids = [122, 117, 120, 115];

// The query will now return different results
```

```

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
   Adams: 120
   Feng: 117
   Garcia: 115
   Tucker: 122
*/

```

Puede usar instrucciones de flujo de control, como `if... else` o `switch`, para seleccionar entre consultas alternativas predeterminadas. En el ejemplo siguiente, `studentQuery` usa una cláusula `where` diferente si el valor del tiempo de ejecución de `oddYear` es `true` o `false`.

C#

```

void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery = oddYear
        ? (from student in students
           where student.Year is GradeLevel.FirstYear or
GradeLevel.ThirdYear
           select student)
        : (from student in students
           where student.Year is GradeLevel.SecondYear or
GradeLevel.FourthYear
           select student);
    var descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year
level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
   The following students are at an odd year level:
   Fakhouri: 116
   Feng: 117
   Garcia: 115
   Mortensen: 113
   Tucker: 119
   Tucker: 122
*/

```

```
/* Output:  
The following students are at an even year level:  
Adams: 120  
Garcia: 114  
Garcia: 118  
O'Donnell: 112  
Omelchenko: 111  
Zabokritski: 121  
*/
```

Controlar valores nulos en expresiones de consulta

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 `null`, o contiene un elemento cuyo valor es `null`, y la consulta no controla los valores `null`, se inicia un elemento `NullReferenceException` cuando se ejecute la consulta.

Se pueden codificar de forma defensiva para evitar una excepción de referencia nula, tal y como se muestra en el ejemplo siguiente:

C#

```
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>`.

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?`:

C#

```
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 };
```

En cada uno de los ejemplos, se usa la palabra clave de consulta `equals`. También puede usar [coincidencia de patrones](#), que incluye patrones para `is null` y `is not null`. Estos patrones no se recomiendan en las consultas de LINQ, ya que es posible que los proveedores de consultas no interpreten correctamente la sintaxis nueva de C#. Un proveedor de consultas es una biblioteca que traduce expresiones de consulta de C# a un formato de datos nativo, como Entity Framework Core. Los proveedores de consultas implementan la interfaz [System.Linq.IQueryProvider](#) para crear orígenes de datos que implementan la interfaz [System.Linq.IQueryable<T>](#).

Controlar excepciones en expresiones de consulta

Es posible llamar a cualquier método en el contexto de una expresión de consulta. En las expresiones de consulta, no llame a cualquier método 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.

En el ejemplo siguiente se muestra cómo mover código de control de excepciones fuera de una expresión de consulta. Esta refactorización solo es posible cuando el método no depende de ninguna variable local de la consulta. Es más fácil tratar las excepciones fuera de la expresión de consulta.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();
```

```

// DO THIS with a datasource that might
// throw an exception.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
{
    // 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());
    }
}

```

En el bloque `catch (InvalidOperationException)` del ejemplo anterior, controle (o no controle) la excepción de la manera adecuada para la aplicación.

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.

El bloque `try` encierra el bucle `foreach`, no la propia consulta. El bucle `foreach` es el punto en el que se ejecuta la consulta. Las excepciones en tiempo de ejecución se producen cuando se ejecuta la consulta. Por lo tanto, deben controlarse en el bucle `foreach`.

C#

```

// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        @"C:\newFolder\" + s;

// 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;

try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
   Processing C:\newFolder\fileA.txt
   Processing C:\newFolder\fileB.txt
   Operation is not valid due to the current state of the object.
*/

```

Recuerde detectar cualquier excepción que espere generar o realice cualquier limpieza necesaria en un bloque `finally`.

Consulte también

- [Tutorial: Escribir consultas en C#](#)
- [where \(cláusula\)](#)
- [Consultas basadas en el estado del entorno de ejecución](#)
- [Nullable<T>](#)
- [Tipos de valores que aceptan valores NULL](#)

Relaciones entre tipos en operaciones de consulta LINQ (C#)

Artículo • 18/12/2023

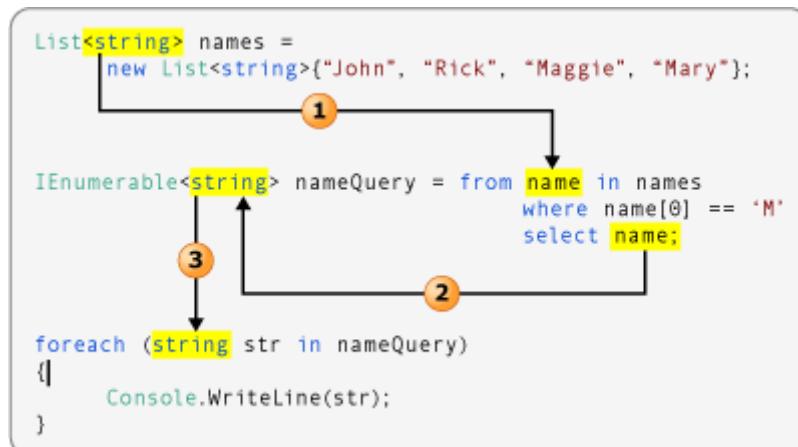
Para escribir las consultas eficazmente, es necesario comprender cómo los tipos de las variables en una operación de consulta completa se relacionan entre sí. Si entiende estas relaciones comprenderá más fácilmente los ejemplos de LINQ y los ejemplos de código de la documentación. Además, comprenderá lo que ocurre cuando las variables se escriben 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 cuando se usa la escritura implícita mediante `var`.

Consultas que no transforman los datos de origen

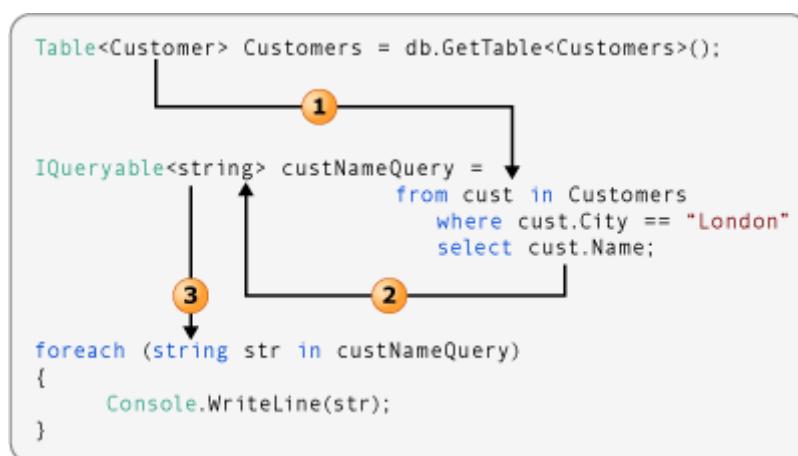
La ilustración siguiente muestra una operación de consulta de LINQ to Objects que no realiza ninguna transformación de los datos. El origen contiene una secuencia de cadenas y el resultado de la consulta también es una secuencia de cadenas.



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 de 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.



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);
}

```

LINQ y tipos genéricos (C#)

Las consultas LINQ se basan en tipos genéricos. 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 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`.

C#

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

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:

C#

```
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](#).

 [Colaborar con nosotros en GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Características de C# compatibles con LINQ

Artículo • 26/04/2024

Expresiones de consulta

Las expresiones de consulta usan una sintaxis declarativa similar a SQL o XQuery para consultar colecciones de `System.Collections.Generic.IEnumerable<T>`. En tiempo de compilación, la sintaxis de consulta se convierte en llamadas de método a la implementación de un proveedor LINQ de los métodos 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.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Variables con asignación implícita de tipos (var)

Use el modificador `var` para indicar al compilador que infiera y asigne el tipo, tal y como se muestra aquí:

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Las variables declaradas como `var` están fuertemente tipadas, como las variables cuyo tipo se especifica explícitamente. El uso de `var` hace posible crear tipos anónimos, pero solo para variables locales. 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:

C#

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuando con la clase `Customer`, suponga que hay un origen de datos denominado `IncomingOrders` y que para cada orden con un `OrderSize` grande, quisiera 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 llenar una colección:

C#

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone =
o.Phone };
```

El origen de datos podría tener más propiedades definidas que la clase `Customer`, como `OrderSize`, pero con la inicialización de objetos, los datos devueltos de la consulta se moldean en el tipo de datos deseado. Usted elige los datos pertinentes para la clase. Como resultado, ahora tiene un `System.Collections.Generic.IEnumerable<T>` lleno con los nuevos `Customer` que quería. El ejemplo anterior también se puede escribir con la sintaxis del método LINQ:

C#

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize >
5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

A partir de C# 12, es posible usar una [expresión de colección](#) para inicializar una colección.

Para más información, vea:

- [Inicializadores de objeto y colección](#)

- Sintaxis de las expresiones de consulta para operadores de consulta estándar

Tipos anónimos

El compilador crea un [tipo anónimo](#). 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:

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

A partir de C# 7, es posible usar [tuplas](#) para crear tipos sin nombre.

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>](#).

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 (la) expresión. En la programación de LINQ, se encuentran expresiones lambda al realizar llamadas de método directas a los operadores de consulta estándar.

Expresiones como datos

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`, devuelve 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.

En el ejemplo siguiente, el primer método `QueryMethod1` devuelve una consulta como un valor devuelto y el segundo método `QueryMethod2` devuelve una consulta como un parámetro `out` (`returnQ` en el ejemplo). En ambos casos, se trata de una consulta que se devuelve, no de los resultados de la consulta.

C#

```
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ =
        from i in ints
        where i < 4
        select i.ToString();

int[] nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

var myQuery1 = QueryMethod1(nums);
```

La consulta `myQuery1` se ejecuta en el siguiente bucle `foreach`.

C#

```
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Reste el puntero del mouse sobre `myQuery1` para ver su tipo.

También puede ejecutar la consulta devuelta desde `QueryMethod1` directamente, sin usar `myQuery1`.

C#

```
foreach (var s in QueryMethod1(nums))
{
```

```
        Console.WriteLine(s);
    }
```

Reste el puntero del mouse sobre la llamada a `QueryMethod1` para ver su tipo de valor devuelto.

`QueryMethod2` devuelve una consulta como el valor de su parámetro `out`:

C#

```
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}
```

Puede modificar una consulta mediante la composición de la consulta. En este caso, el objeto de consulta anterior se usa para crear un nuevo objeto de consulta. Este nuevo objeto devolverá resultados diferentes a los del objeto de consulta original.

C#

```
myQuery1 =
    from item in myQuery1
    orderby item descending
    select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

Abrir incidencia con la documentación

nuestra guía para
colaboradores.

☒ Proporcionar comentarios sobre
el producto

Tutorial: Write queries in C# using language integrated query (LINQ)

Article • 04/25/2024

In this tutorial, you create a data source and write several LINQ queries. You can experiment with the query expressions and see the differences in the results. This walkthrough demonstrates the C# language features that are used to write LINQ query expressions. You can follow along and build the app and experiment with the queries yourself. This article assumes you've installed the latest .NET SDK. If not, go to the [.NET Downloads page](#) and install the latest version on your machine.

First, create the application. From the console, type the following command:

.NET CLI

```
dotnet new console -o WalkthroughWritingLinqQueries
```

Or, if you prefer Visual Studio, create a new console application named *WalkthroughWritingLinqQueries*.

Create an in-memory data source

The first step is to create a data source for your queries. The data source for the queries is a simple list of `Student` records. Each `Student` record has a first name, family name, and an array of integers that represents their test scores in the class. Add a new file named *students.cs*, and copy the following code into that file:

C#

```
namespace WalkthroughWritingLinqQueries;

public record Student(string First, string Last, int ID, int[] Scores);
```

Note the following characteristics:

- The `Student` record consists of autoimplemented properties.
- Each student in the list is initialized with the primary constructor.
- The sequence of scores for each student is initialized with a primary constructor.

Next, create a sequence of `Student` records that serves as the source of this query. Open *Program.cs*, and remove the following boilerplate code:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Replace it with the following code that creates a sequence of `Student` records:

C#

```
using WalkthroughWritingLinqQueries;

// Create a data source by using a collection initializer.
IEnumerable<Student> students =
[
    new Student(First: "Svetlana", Last: "Omelchenko", ID: 111, Scores: [97, 92, 81, 60]),
    new Student(First: "Claire", Last: "O'Donnell", ID: 112, Scores: [75, 84, 91, 39]),
    new Student(First: "Sven", Last: "Mortensen", ID: 113, Scores: [88, 94, 65, 91]),
    new Student(First: "Cesar", Last: "Garcia", ID: 114, Scores: [97, 89, 85, 82]),
    new Student(First: "Debra", Last: "Garcia", ID: 115, Scores: [35, 72, 91, 70]),
    new Student(First: "Fadi", Last: "Fakhouri", ID: 116, Scores: [99, 86, 90, 94]),
    new Student(First: "Hanying", Last: "Feng", ID: 117, Scores: [93, 92, 80, 87]),
    new Student(First: "Hugo", Last: "Garcia", ID: 118, Scores: [92, 90, 83, 78]),

    new Student("Lance", "Tucker", 119, [68, 79, 88, 92]),
    new Student("Terry", "Adams", 120, [99, 82, 81, 79]),
    new Student("Eugene", "Zabokritski", 121, [96, 85, 91, 60]),
    new Student("Michael", "Tucker", 122, [94, 92, 91, 91])
];
```

- The sequence of students is initialized with a collection expression.
- The `Student` record type holds the static list of all students.
- Some of the constructor calls use `named arguments` to clarify which argument matches which constructor parameter.

Try adding a few more students with different test scores to the list of students to get more familiar with the code so far.

Create the query

Next, you create your first query. Your query, when you execute it, produces a list of all students whose score on the first test was greater than 90. Because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the `var` keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see [Implicitly Typed Local Variables](#).) Add the following code to `Program.cs`, after the code that creates the sequence of students:

C#

```
// 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;
```

The query's range variable, `student`, serves as a reference to each `Student` in the source, providing member access for each object.

Run the query

Now write the `foreach` loop that causes the query to execute. Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop. The type of this variable is `Student`, and the type of the query variable is compatible, `IEnumerable<Student>`. After you added the following code, build and run the application to see the results in the **Console** window.

C#

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine($"{student.Last}, {student.First}");  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry
```

```
// Zabokritski, Eugene  
// Tucker, Michael
```

To further refine the query, you can combine multiple Boolean conditions in the `where` clause. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

C#

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Try the preceding `where` clause, or experiment yourself with other filter conditions. For more information, see [where clause](#).

Order the query results

It's easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the family name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

C#

```
orderby student.Last ascending
```

Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

C#

```
orderby student.Scores[0] descending
```

Change the `WriteLine` format string so that you can see the scores:

C#

```
Console.WriteLine($"{student.Last}, {student.First} {student.Scores[0]}");
```

For more information, see [orderby clause](#).

Group the results

Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their family name as the key.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =  
    from student in students  
    group student by student.Last[0];
```

The type of the query changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `student` objects. The code in the `foreach` execution loop also must change:

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery)  
{  
    Console.WriteLine(studentGroup.Key);  
    foreach (Student student in studentGroup)  
    {  
        Console.WriteLine($"    {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
```

Run the application and view the results in the **Console** window. For more information, see [group clause](#).

Explicitly coding `IEnumerables` of `IGroupings` can quickly become tedious. Write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword doesn't change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. In the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works as before. Change the `student` iteration variable to `var` and run the query again. You see that you get exactly the same results.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];

foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"    {student.Last}, {student.First}");
    }
}
```

For more information about `var`, see [Implicitly Typed Local Variables](#).

Order the groups by their key value

The groups in the previous query aren't in alphabetical order. You can provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:

C#

```
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($"    {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

```

Run this query, and the groups are now sorted in alphabetical order.

You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example. It can also enhance performance by storing the results of an expression so that it doesn't have to be calculated multiple times.

C#

```

// 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
```

For more information, see the article on the [let clause](#).

Use method syntax in a query expression

As described in [Query Syntax and Method Syntax in LINQ](#), some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

C#

```
var studentQuery =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

To transform or project in the select clause

It's common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. The query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.

C#

```
IEnumerable<string> studentQuery =
    from student in students
    where student.Last == "Garcia"
```

```
select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `Students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

C#

```
var aboveAverageQuery =
    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 aboveAverageQuery)
{
    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
```

Standard Query Operators Overview

Article • 05/31/2024

The *standard query operators* are the keywords and methods that form the LINQ pattern. The C# language defines [LINQ query keywords](#) that you use for the most common query expression. The compiler translates expressions using these keywords to the equivalent method calls. The two forms are synonymous. Other methods that are part of the [System.Linq](#) namespace don't have equivalent query keywords. In those cases, you must use the method syntax. This section covers all the query operator keywords. The runtime and other NuGet packages add more methods designed to work with LINQ queries each release. The most common methods, including those that have query keyword equivalents are covered in this section. For the full list of query methods supported by the .NET Runtime, see the [System.Linq.Enumerable](#) API documentation. In addition to the methods covered here, this class contains methods for concatenating data sources, computing a single value from a data source, such as a sum, average, or other value.

Important

These samples use an [System.Collections.Generic.IEnumerable<T>](#) data source. Data sources based on [System.Linq.IQueryProvider](#) use [System.Linq.IQueryable<T>](#) data sources and [expression trees](#). Expression trees have [limitations](#) on the allowed C# syntax. Furthermore, each [IQueryProvider](#) data source, such as [EF Core](#) may impose more restrictions. Check the documentation for your data source.

Most of these methods operate on sequences, where a sequence is an object whose type implements the [IEnumerable<T>](#) interface or the [IQueryable<T>](#) interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting and more. The methods that make up each set are static members of the [Enumerable](#) and [Queryable](#) classes, respectively. They're defined as [extension methods](#) of the type that they operate on.

The distinction between [IEnumerable<T>](#) and [IQueryable<T>](#) sequences determines how the query is executed at runtime.

For [IEnumerable<T>](#), the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

For `IQueryable<T>`, the query is translated into an [expression tree](#). The expression tree can be translated to a native query when the data source can optimize the query. Libraries such as [Entity Framework](#) translate LINQ queries into native SQL queries that execute at the database.

The following code example demonstrates how the standard query operators can be used to obtain information about a sequence.

C#

```
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
```

Where possible, the queries in this section use a sequence of words or numbers as the input source. For queries where more complicated relationships between objects are used, the following sources that model a school are used:

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `city` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head.

You can find the data set in the [source repo ↗](#).

Types of query operators

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that

return a singleton value (such as [Average](#) and [Sum](#)) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object. You can use the output sequence of one query as the input sequence to another query. Calls to query methods can be chained together in one query, which enables queries to become arbitrarily complex.

Query operators

In a LINQ query, the first step is to specify the data source. In a LINQ query, the `from` clause comes first in order to introduce the data source (`students`) and the *range variable* (`student`).

C#

```
//queryAllStudents is an IEnumerable<Student>
var queryAllStudents = from student in students
                        select student;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable serves as a reference to each successive element in `students`. Because the compiler can infer the type of `student`, you don't have to specify it explicitly. You can introduce more range variables in a `let` clause. For more information, see [let clause](#).

ⓘ Note

For non-generic data sources such as [ArrayList](#), the range variable must be explicitly typed. For more information, see [How to query an ArrayList with LINQ \(C#\) and from clause](#).

Once you obtain a data source, you can perform any number of operations on that data source:

- [Filter data](#) using the `where` keyword.
- [Order data](#) using the `orderby` and optionally `descending` keywords.
- [Group data](#) using the `group` and optionally `into` keywords.
- [Join data](#) using the `join` keyword.
- [Project data](#) using the `select` keyword.

Query Expression Syntax Table

The following table lists the standard query operators that have equivalent query expression clauses.

[Expand table](#)

Method	C# query expression syntax
Cast	Use an explicitly typed range variable: <code>from int i in numbers</code> (For more information, see from clause .)
GroupBy	<code>group ... by</code> -or- <code>group ... by ... into ...</code> (For more information, see group clause .)
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	<code>join ... in ... on ... equals ... into ...</code> (For more information, see join clause .)
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	<code>join ... in ... on ... equals ...</code> (For more information, see join clause .)
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	<code>orderby</code> (For more information, see orderby clause .)

Method	C# query expression syntax
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ... descending</pre> <p>(For more information, see orderby clause.)</p>
Select	<pre>select</pre> <p>(For more information, see select clause.)</p>
SelectMany	<p>Multiple <code>from</code> clauses.</p> <p>(For more information, see from clause.)</p>
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ..., ...</pre> <p>(For more information, see orderby clause.)</p>
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ..., ... descending</pre> <p>(For more information, see orderby clause.)</p>
Where	<pre>where</pre> <p>(For more information, see where clause.)</p>

Data Transformations with LINQ

Language-Integrated Query (LINQ) isn't only about retrieving data. It's also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping.

But perhaps the most powerful feature of LINQ queries is the ability to create new types. The `select` clause creates an output element from an input element. You use it to transform an input element into an output element:

- Merge multiple input sequences into a single output sequence that has a new type.
- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

These transformations can be combined in various ways in the same query. Furthermore, the output sequence of one query can be used as the input sequence for a new query.

The following example transforms objects in an in-memory data structure into XML elements.

C#

```
// 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.FirstName),  
        new XElement("Last", student.LastName),  
        new XElement("Scores", scores)  
    ) // end "student"  
); // end "Root"  
  
// Execute the query.  
Console.WriteLine(studentsToXML);
```

The code produces the following XML output:

XML

```
<Root>  
  <student>  
    <First>Svetlana</First>  
    <Last>Omelchenko</Last>  
    <Scores>97,90,73,54</Scores>  
  </student>  
  <student>  
    <First>Claire</First>  
    <Last>O'Donnell</Last>
```

```

<Scores>56,78,95,95</Scores>
</student>
...
<student>
<First>Max</First>
<Last>Lindgren</Last>
<Scores>86,88,96,63</Scores>
</student>
<student>
<First>Arina</First>
<Last>Ivanova</Last>
<Scores>93,63,70,80</Scores>
</student>
</Root>

```

For more information, see [Creating XML Trees in C# \(LINQ to XML\)](#).

You can use the results of one query as the data source for a subsequent query. This example shows how to order the results of a join operation. This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a subquery orders all the matching elements from the products sequence.

C#

```

var orderedQuery = from department in departments
                   join student in students on department.ID equals
                   student.DepartmentID into studentGroup
                   orderby department.Name
                   select new
                   {
                       DepartmentName = department.Name,
                       Students = from student in studentGroup
                                   orderby student.LastName
                                   select student
                   };

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($" {student.LastName,-10}{student.FirstName,-10}");
    }
}
/* Output:
Chemistry
Balzan    Josephine
Fakhouri   Fadi
Popov     Innocenty
Seleznyova Sofiya

```

Vella	Carmen
Economics	
Adams	Terry
Adaobi	Izuchukwu
Berggren	Jeanette
Garcia	Cesar
Ifeoma	Nwanneka
Jamuike	Ifeanacho
Larsson	Naima
Svensson	Noel
Ugomma	Ifunanya
Engineering	
Axelsson	Erik
Berg	Veronika
Engström	Nancy
Hicks	Cassie
Keever	Bruce
Micallef	Nicholas
Mortensen	Sven
Nilsson	Erna
Tucker	Michael
Yermolayeva Anna	
English	
Andersson	Sarah
Feng	Hanying
Ivanova	Arina
Jakobsson	Jesper
Jensen	Christiane
Johansson	Mark
Kolpakova	Nadezhda
Omelchenko	Svetlana
Urquhart	Donald
Mathematics	
Frost	Gaby
Garcia	Hugo
Hedlund	Anna
Kovaleva	Katerina
Lindgren	Max
Maslova	Evgeniya
Olsson	Ruth
Sammut	Maria
Sazonova	Anastasiya
Physics	
Åkesson	Sami
Edwards	Amy E.
Falzon	John
Garcia	Debra
Hansson	Sanna
Mattsson	Martina
Richardson	Don
Zabokritski	Eugene

*/

The equivalent query using method syntax is shown in the following code:

C#

```
var orderedQuery = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, studentGroup) => new
    {
        DepartmentName = department.Name,
        Students = studentGroup.OrderBy(student => student.LastName)
    })
    .OrderBy(department => department.DepartmentName);

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($" {student.LastName,-10}
{student.FirstName,-10}");
    }
}
```

Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we don't recommend it. Some LINQ providers might not preserve that ordering after the join. For more information, see [join clause](#).

See also

- [Enumerable](#)
- [Queryable](#)
- [select clause](#)
- [Extension Methods](#)
- [Query Keywords \(LINQ\)](#)
- [Anonymous Types](#)

Filtrar datos en C# con LINQ

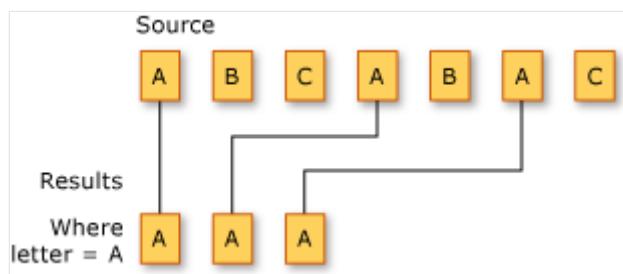
Artículo • 03/06/2024

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 *seleccionar* elementos que coinciden con la condición especificada.

ⓘ Importante

Estos ejemplos usan un origen de datos de [System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan orígenes de datos de [System.Linq.IQueryable<T>](#) y [árboles de expresiones](#). Los árboles de expresiones tienen [limitaciones](#) en la sintaxis permitida de C#. Además, cada origen de datos de [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

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.

[Expandir tabla](#)

Nombre del método	Descripción	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 es aplicable.	Enumerable.OfType Queryable.OfType

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
Where	Selecciona valores basados en una función de predicado.	where	Enumerable.Where Queryable.Where

En el siguiente ejemplo se usa la cláusula `where` para filtrar de una matriz aquellas cadenas que tienen una longitud específica.

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IQueryable<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
*/
```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IQueryable<string> query =
    words.Where(word => word.Length == 3);

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
```

Consulte también

- [System.Linq](#)
- [where \(cláusula\)](#)
- [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 \(LINQ\) \(C#\)](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Operaciones de proyección (C#)

Artículo • 04/06/2024

El término "proyección" hace referencia a la operación de transformar un objeto en una nueva forma que, a menudo, consta solo de aquellas propiedades usadas 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.

ⓘ Importante

Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan [System.Linq.IQueryable<T>](#) orígenes de datos y [árboles de expresiones](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos de [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

Los métodos del operador de consulta estándar que realizan proyecciones se indican en la sección siguiente.

Métodos

[] Expandir tabla

Nombres de método	Descripción	Sintaxis de la expresión de consulta de C#	Información adicional
Seleccionar	Proyecta valores basados en una función de transformación.	<code>select</code>	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 <code>from</code>	Enumerable.SelectMany Queryable.SelectMany
Zip	Genera una secuencia de tuplas con elementos a partir de dos o tres secuencias especificadas.	No aplicable.	Enumerable.Zip Queryable.Zip

Select

En el ejemplo siguiente se usa la cláusula `select` para proyectar la primera letra de cada cadena de una lista de cadenas.

C#

```
List<string> words = ["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  
*/
```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```
List<string> words = ["an", "apple", "a", "day"];  
  
var query = words.Select(word => 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.

C#

```
List<string> phrases = ["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  
*/
```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];  
  
var query = phrases.SelectMany(phrases => phrases.Split(' '));  
  
foreach (string s in query)  
{  
    Console.WriteLine(s);  
}  
  
/* This code produces the following output:  
  
an  
apple  
a  
day  
the  
quick  
brown
```

```
    fox  
*/
```

El método `SelectMany` también puede formar la combinación de hacer coincidir todos los elementos de la primera secuencia con cada elemento de la segunda secuencia:

```
C#
```

```
var query = from number in numbers  
            from letter in letters  
            select (number, letter);  
  
foreach (var item in query)  
{  
    Console.WriteLine(item);  
}
```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

```
C#
```

```
var method = numbers  
.SelectMany(number => letters,  
(number, letter) => (number, letter));  
  
foreach (var item in method)  
{  
    Console.WriteLine(item);  
}
```

Zip

Hay varias sobrecargas para el operador `Zip` de proyección. Todos los métodos `Zip` funcionan en secuencias de dos o más tipos posiblemente heterogéneos. Las dos primeras sobrecargas devuelven tuplas, con el tipo posicional correspondiente de las secuencias dadas.

Observe las siguientes colecciones:

```
C#
```

```
// An int array with 7 elements.  
IEnumerable<int> numbers = [1, 2, 3, 4, 5, 6, 7];
```

```
// A char array with 6 elements.  
IEnumerable<char> letters = [ 'A', 'B', 'C', 'D', 'E', 'F'];
```

Para proyectar estas secuencias juntas, use el operador `Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)`:

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))  
{  
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");  
}  
// This code produces the following output:  
//      Number: 1 zipped with letter: 'A'  
//      Number: 2 zipped with letter: 'B'  
//      Number: 3 zipped with letter: 'C'  
//      Number: 4 zipped with letter: 'D'  
//      Number: 5 zipped with letter: 'E'  
//      Number: 6 zipped with letter: 'F'
```

ⓘ Importante

La secuencia resultante de una operación zip nunca tiene más longitud que la secuencia más corta. Las colecciones `numbers` y `letters` difieren en longitud, y la secuencia resultante omite el último elemento de la colección `numbers`, ya que no tiene nada con que comprimir.

La segunda sobrecarga acepta una secuencia `third`. Vamos a crear otra colección, concretamente `emoji`:

C#

```
// A string array with 8 elements.  
IEnumerable<string> emoji = [ "😊", "🔥", "🎉", "👀", "⭐", "❤️", "✓",  
"💯"];
```

Para proyectar estas secuencias juntas, use el operador `Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)`:

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters,  
emoji))  
{
```

```

        Console.WriteLine(
            $"Number: {number} is zipped with letter: '{letter}' and emoji:
{em}");
}
// This code produces the following output:
//    Number: 1 is zipped with letter: 'A' and emoji: 😊
//    Number: 2 is zipped with letter: 'B' and emoji: 🔥
//    Number: 3 is zipped with letter: 'C' and emoji: 🎉
//    Number: 4 is zipped with letter: 'D' and emoji: 💡
//    Number: 5 is zipped with letter: 'E' and emoji: ⭐
//    Number: 6 is zipped with letter: 'F' and emoji: 💜

```

Al igual que la sobrecarga anterior, el método `Zip` proyecta una tupla, pero esta vez con tres elementos.

La tercera sobrecarga acepta un argumento `Func<TFirst, TSecond, TResult>` que actúa como selector de resultados. Puede proyectar una nueva secuencia resultante de las secuencias que se comprimen.

C#

```

foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({int}letter)"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//    1 = A (65)
//    2 = B (66)
//    3 = C (67)
//    4 = D (68)
//    5 = E (69)
//    6 = F (70)

```

Con la sobrecarga `Zip` anterior, la función especificada se aplica a los elementos correspondientes `numbers` y `letter`, lo que genera una secuencia de los resultados `string`.

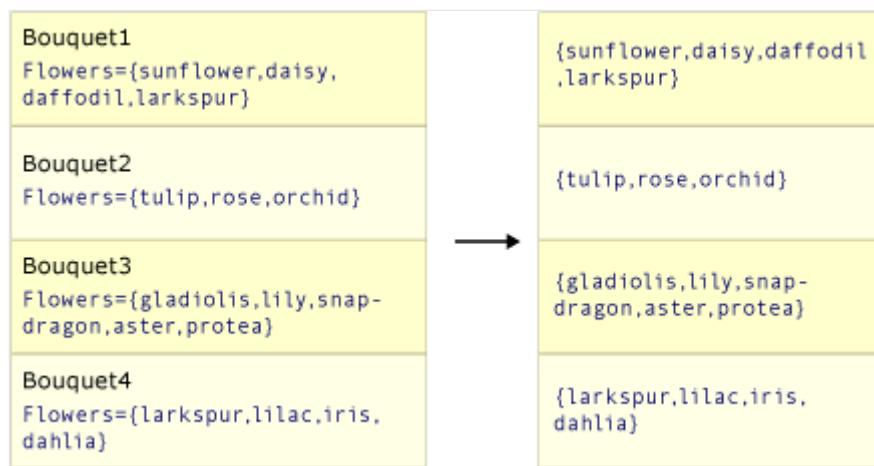
Diferencias entre `Select` y `SelectMany`

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

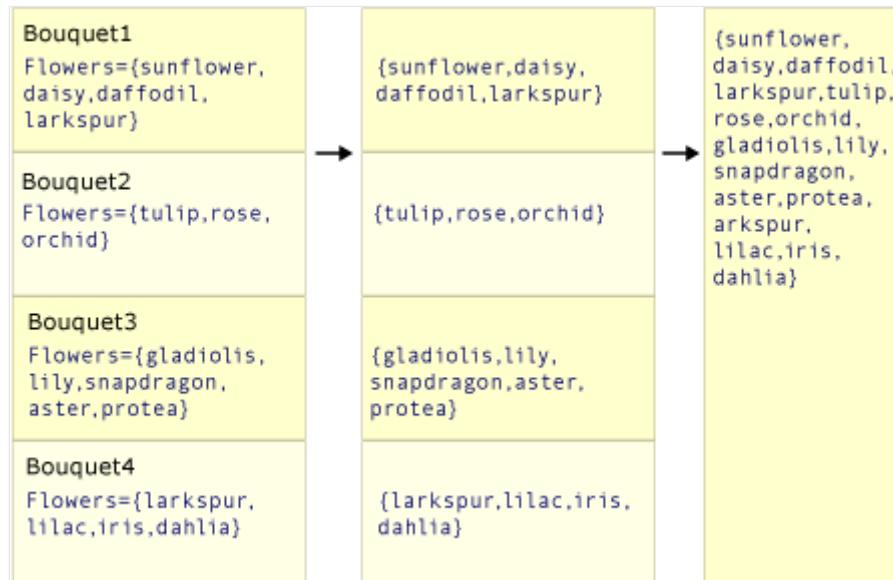
valor de origen. La función de transformación que se pasa como argumento a `SelectMany` debe devolver una secuencia enumerable de valores para cada valor de origen. `SelectMany` concatena estas secuencias enumerables para crear una secuencia grande.

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 elementos de cada lista de nombres de flores de la colección de origen. En el siguiente ejemplo, el "valor único" que usa la función de transformación `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)` es una colección de valores. Este ejemplo requiere el bucle adicional `foreach` a fin de enumerar cada una de las cadenas de cada subsecuencia.

```
C#  
  
class Bouquet  
{  
    public required List<string> Flowers { get; init; }  
}  
  
static void SelectVsSelectMany()  
{  
    List<Bouquet> bouquets =  
    [  
        new Bouquet { Flowers = ["sunflower", "daisy", "daffodil",  
"larkspur"] },  
        new Bouquet { Flowers = ["tulip", "rose", "orchid"] },  
        new Bouquet { Flowers = ["gladiolus", "lily", "snapdragon", "aster",  
"protea"] },  
        new Bouquet { Flowers = ["larkspur", "lilac", "iris", "dahlia"] }  
    ];  
  
    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);  
  
    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);  
    }  
}
```

Consulte también

- System.Linq
- select (cláusula)
- Procedimiento para llenar colecciones de objetos de varios orígenes (LINQ) (C#)
- Procedimiento para dividir un archivo en varios mediante el uso de grupos (LINQ) (C#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Operaciones set [C#]

Artículo • 04/06/2024

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.

ⓘ Importante

Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan [System.Linq.IQueryable<T>](#) orígenes de datos y [árboles de expresión](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

 Expandir tabla

Nombres de método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
<code>Distinct</code> o <code>DistinctBy</code>	Quita valores duplicados de una colección.	No aplicable.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
<code>Except</code> o <code>ExceptBy</code>	Devuelve la diferencia de conjuntos, es decir, los elementos de una colección que no aparecen en una segunda colección.	No aplicable.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
<code>Intersect</code> o <code>IntersectBy</code>	Devuelve la intersección de conjuntos, es decir, los elementos que aparecen en las dos colecciones.	No aplicable.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
<code>Union</code> o <code>UnionBy</code>	Devuelve la unión de conjuntos, es decir, los elementos únicos que aparecen en una de las dos colecciones.	No es aplicable.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

Distinct y DistinctBy

En la siguiente ilustración se muestra el comportamiento del método `Enumerable.Distinct` en una secuencia de cadenas. La secuencia devuelta contiene los elementos únicos de la secuencia de entrada.



C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

IEnumerable<string> query = from word in words.Distinct()
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */
```

`DistinctBy` es un enfoque alternativo a `Distinct`, que adopta `keySelector`. `keySelector` se usa como discriminador comparativo del tipo de origen. En el siguiente código, las palabras se discriminan en función de su `Length`, y se muestra la primera palabra de cada longitud:

C#

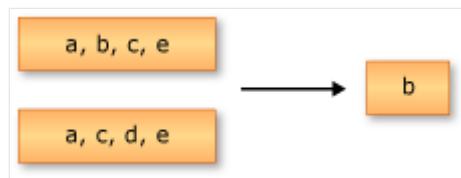
```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

foreach (string word in words.DistinctBy(p => p.Length))
{
    Console.WriteLine(word);
}
```

```
// This code produces the following output:  
//     the  
//     quick  
//     jumped  
//     over
```

Except y ExceptBy

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.



ⓘ Nota

En los ejemplos siguientes de este artículo se usan los orígenes de datos comunes para esta área.

Cada `Student` tiene un nivel académico, un departamento principal y una serie de puntuaciones. Un `Teacher` también tiene una propiedad `City` que identifica el campus donde el profesor imparte clases. Un `Department` tiene un nombre y una referencia a un `Teacher` que actúa como jefe del departamento.

Puede encontrar el conjunto de datos de ejemplo en el repositorio [de ↗ origen](#).

C#

```
public enum GradeLevel  
{  
    FirstYear = 1,  
    SecondYear,  
    ThirdYear,  
    FourthYear  
};  
  
public class Student  
{  
    public required string FirstName { get; init; }  
    public required string LastName { get; init; }  
    public required int ID { get; init; }  
  
    public required GradeLevel Year { get; init; }  
    public required List<int> Scores { get; init; }  
}
```

```

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}

```

C#

```

string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Except(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * quick
 * brown
 * fox
 */

```

El método `ExceptBy` es un enfoque alternativo a `Except` que adopta dos secuencias de tipos posiblemente heterogéneos y `keySelector`. El `keySelector` es el mismo tipo que el tipo de la primera colección. Tenga en cuenta la siguiente matriz `Teacher` e identificadores de profesor que se van a excluir. Para buscar profesores en la primera colección que no están en la segunda, puede proyectar el identificador de profesor en la segunda colección:

C#

```

int[] teachersToExclude =
[
    901,    // English
    965,    // Mathematics
    932,    // Engineering
    945,    // Economics
    987,    // Physics
    901     // Chemistry
];

foreach (Teacher teacher in
    teachers.ExceptBy(
        teachersToExclude, teacher => teacher.ID))
{
    Console.WriteLine($"{teacher.First} {teacher.Last}");
}

```

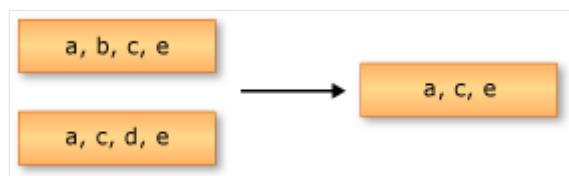
En el código de C# anterior:

- La matriz `teachers` se filtra solo a los profesores que no están en la matriz `teachersToExclude`.
- La matriz `teachersToExclude` contiene el valor `ID` de todos los jefes de departamento.
- La llamada a `ExceptBy` da como resultado un nuevo conjunto de valores que se escriben en la consola.

El nuevo conjunto de valores es de tipo `Teacher`, que es el tipo de la primera colección. Cada `teacher` de la matriz `teachers` que no tiene un valor de identificador correspondiente en la matriz `teachersToExclude` se escribe en la consola.

Intersect y IntersectBy

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.



C#

```

string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

```

```

IEnumerable<string> query = from word in words1.Intersect(words2)
                             select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 */

```

El método `IntersectBy` es un enfoque alternativo a `Intersect` que adopta dos secuencias de tipos posiblemente heterogéneos y `keySelector`. `keySelector` se usa como discriminador comparativo del tipo de la segunda colección. Tenga en cuenta las siguientes matrices de estudiantes y profesores. La consulta coincide con los elementos de cada secuencia por nombre para buscar a los alumnos que también son profesores:

C#

```

foreach (Student person in
         students.IntersectBy(
             teachers.Select(t => (t.First, t.Last)), s => (s.FirstName,
s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}

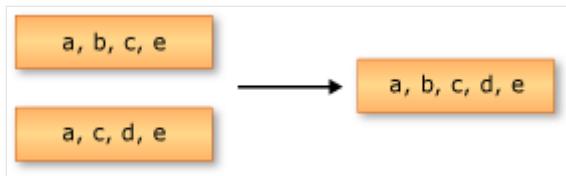
```

En el código de C# anterior:

- La consulta genera la intersección de `Teacher` y `Student` comparando nombres.
- Solo las personas que se encuentran en ambas matrices están presentes en la secuencia resultante.
- Las instancias `Student` resultantes se escriben en la consola.

Union y UnionBy

En el siguiente ejemplo se muestra una operación de unión en dos secuencias de cadenas. La secuencia devuelta contiene los elementos únicos de las dos secuencias de entrada.



C#

```

string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IQueryable<string> query = from word in words1.Union(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */

```

[UnionBy](#) es un enfoque alternativo a `Union` que adopta dos secuencias del mismo tipo y `keySelector`. `keySelector` se usa como discriminador comparativo del tipo de origen. La siguiente consulta genera la lista de todas las personas que son estudiantes o profesores. Los estudiantes que también son profesores se agregan a la unión establecida una sola vez:

C#

```

foreach (var person in
    students.Select(s => (s.FirstName, s.LastName)).UnionBy(
        teachers.Select(t => (FirstName: t.First, LastName: t.Last)), s =>
    (s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}

```

En el código de C# anterior:

- Las matrices `teachers` y `students` se entrelazan con sus nombres como selector de claves.
- Los nombres resultantes se escriben en la consola.

Consulte también

- [System.Linq](#)
- [Procedimiento para buscar la diferencia de conjuntos entre dos listas \(LINQ\) \(C#\)](#)

Ordenación de datos (C#)

Artículo • 06/06/2024

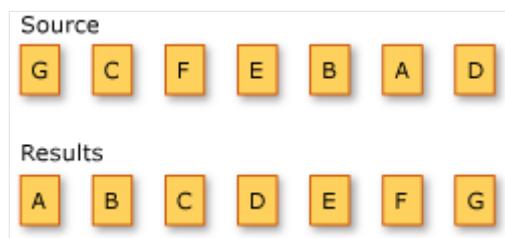
Una operación de ordenación ordena los elementos de una secuencia según uno o varios atributos. El primer criterio de ordenación realiza una ordenación primaria de los elementos. Al especificar un segundo criterio de ordenación, se pueden ordenar los elementos dentro de cada grupo de ordenación primaria.

ⓘ Importante

Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan [System.Linq.IQueryable<T>](#) orígenes de datos y [árboles de expresión](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

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

[Expandir tabla](#)

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
OrderBy	Ordena valores en orden ascendente.	<code>orderby</code>	Enumerable.OrderBy Queryable.OrderBy

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
OrderByDescending	Ordena valores en orden descendente.	orderby ... descending	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Realiza una ordenación secundaria en orden ascendente.	orderby ..., ... descending	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 aplicable.	Enumerable.Reverse Queryable.Reverse

ⓘ Nota

En los ejemplos siguientes de este artículo se usan los orígenes de datos comunes para esta área.

Cada `Student` tiene un nivel académico, un departamento principal y una serie de puntuaciones. Un `Teacher` también tiene una propiedad `City` que identifica el campus donde el profesor imparte clases. Un `Department` tiene un nombre y una referencia a un `Teacher` que actúa como jefe del departamento.

Puede encontrar el conjunto de datos de ejemplo en el repositorio [de](#) origen.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }
```

```

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}

```

Orden ascendente principal

El siguiente ejemplo muestra cómo usar la cláusula `orderby` en una consulta LINQ para ordenar la matriz de profesores por apellido, en orden ascendente.

C#

```

IEnumerable<string> query = from teacher in teachers
                               orderby teacher.Last
                               select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}

```

La consulta equivalente escrita usando la sintaxis del método se muestra en el siguiente código:

C#

```

IEnumerable<string> query = teachers
    .OrderBy(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)

```

```
{  
    Console.WriteLine(str);  
}
```

Orden descendente principal

El siguiente ejemplo muestra cómo usar la cláusula `orderby descending` en una consulta LINQ para ordenar a los profesores por apellido, en orden descendente.

C#

```
IEquatable<string> query = from teacher in teachers  
                           orderby teacher.Last descending  
                           select teacher.Last;  
  
foreach (string str in query)  
{  
    Console.WriteLine(str);  
}
```

La consulta equivalente escrita usando la sintaxis del método se muestra en el siguiente código:

C#

```
IEnumerable<string> query = teachers
    .OrderByDescending(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

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. Los profesores se ordenan principalmente por ciudad y en segundo lugar por su apellido, ambos en orden ascendente.

C#

```
        select (teacher.Last, teacher.City);

    foreach ((string last, string city) in query)
    {
        Console.WriteLine($"City: {city}, Last Name: {last}");
    }
```

La consulta equivalente escrita usando la sintaxis del método se muestra en el siguiente código:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenBy(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

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. Los profesores se ordenan principalmente por ciudad y en segundo lugar por su apellido.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                                         orderby teacher.City, teacher.Last descending
                                         select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

La consulta equivalente escrita usando la sintaxis del método se muestra en el siguiente código:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
```

```
.ThenByDescending(teacher => teacher.Last)
.Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Consulte también

- [System.Linq](#)
- [orderby \(cláusula\)](#)
- [Procedimiento para ordenar o filtrar datos de texto por palabra o campo \(LINQ\) \(C#\)](#)

Operaciones cuantificadoras en LINQ (C#)

Artículo • 04/06/2024

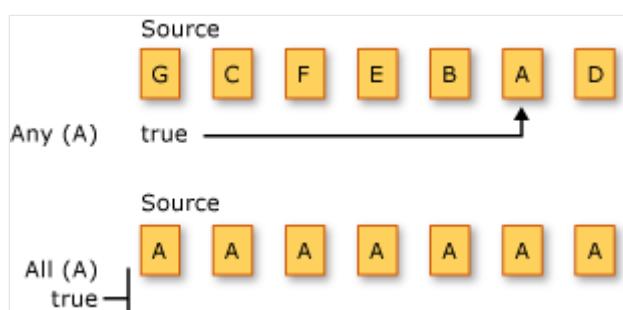
Las operaciones cuantificadoras devuelven un valor [Boolean](#) que indica si algunos o todos los elementos de una secuencia cumplen una condición.

ⓘ Importante

En estos ejemplos se usa un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan orígenes de datos [System.Linq.IQueryable<T>](#) y [árboles de expresión](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos de [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

En la siguiente ilustración se muestran dos operaciones cuantificadoras diferentes en dos secuencias de origen distintas. La primera operación pregunta si alguno de los elementos es el carácter "A". La segunda operación pregunta si todos los elementos son el carácter "A". Ambos métodos devuelven `true` en este ejemplo.



[Expandir tabla](#)

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
Todas	Determina si todos los elementos de una secuencia cumplen una condición.	No es aplicable.	Enumerable.All Queryable.All
Cualquiera	Determina si algunos de los elementos de una secuencia	No es aplicable.	Enumerable.Any Queryable.Any

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
	cumplen una condición.		
Contiene	Determina si una secuencia contiene un elemento especificado.	No aplicable.	Enumerable.Contains Queryable.Contains

All

En el siguiente ejemplo se usa el `All` para encontrar a los estudiantes que han obtenido una puntuación por encima de 70 en todos los exámenes.

```
C#
IEnumerable<string> names = from student in students
                               where student.Scores.All(score => score > 70)
                               select $"{student.FirstName} {student.LastName}";
{string.Join(", ", student.Scores.Select(s => s.ToString()))};

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Cesar Garcia: 71, 86, 77, 97
// Nancy Engström: 75, 73, 78, 83
// Ifunanya Ugomma: 84, 82, 96, 80
```

Any

En el siguiente ejemplo se usa el `Any` para encontrar a los estudiantes que han obtenido una puntuación por encima de 95 en cualquier examen.

```
C#
IEnumerable<string> names = from student in students
                               where student.Scores.Any(score => score > 95)
                               select $"{student.FirstName} {student.LastName}";
{student.Scores.Max()}";

foreach (string name in names)
{
```

```
        Console.WriteLine($"'{name}'");
    }

    // This code produces the following output:
    //
    // Svetlana Omelchenko: 97
    // Cesar Garcia: 97
    // Debra Garcia: 96
    // Ifeanacho Jamuike: 98
    // Ifunanya Ugomma: 96
    // Michelle Caruana: 97
    // Nwanneka Ifeoma: 98
    // Martina Mattsson: 96
    // Anastasiya Sazonova: 96
    // Jesper Jakobsson: 98
    // Max Lindgren: 96
```

Contiene

En el siguiente ejemplo se usa el `Contains` para encontrar a los estudiantes que han obtenido una puntuación de exactamente 95 en un examen.

```
C#
```

```
IEnumerable<string> names = from student in students
                               where student.Scores.Contains(95)
                               select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"'{name}'");
}

// This code produces the following output:
//
// Claire O'Donnell: 56, 78, 95, 95
// Donald Urquhart: 92, 90, 95, 57
```

Consulte también

- [System.Linq](#)
- [Especificación de filtros con predicado de forma dinámica en tiempo de ejecución](#)
- [Procedimiento para buscar frases que contengan un conjunto especificado de palabras \(LINQ\) \(C#\)](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Realizar particiones de datos (C#)

Artículo • 03/06/2024

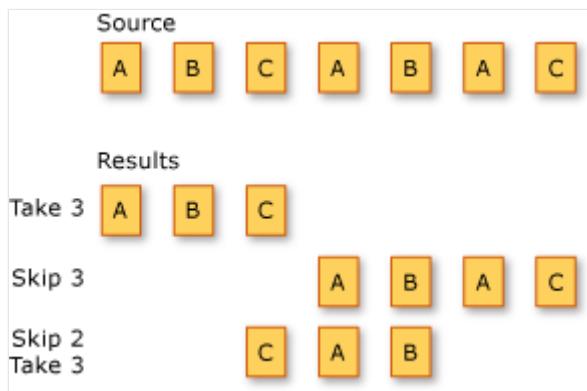
Partición en LINQ es la operación de dividir una secuencia de entrada en dos secciones, sin reorganizar los elementos, y devolver después una de las secciones.

ⓘ Importante

Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan orígenes de datos de [System.Linq.IQueryable<T>](#) y [árboles de expresiones](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos de [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

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

 Expandir tabla

Nombres de método	Descripción	Sintaxis de la expresión de consulta de C#	Información adicional
Skip	Omite los elementos hasta una determinada posición de una secuencia.	No es aplicable.	Enumerable.Skip Queryable.Skip
SkipWhile	Omite los elementos según una función de predicado hasta que un elemento no satisface la condición.	No aplicable.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Admite los elementos hasta una determinada posición de una secuencia.	No es aplicable.	Enumerable.Take Queryable.Take
TakeWhile	Toma los elementos según una función de predicado hasta que un elemento no satisface la condición.	No aplicable.	Enumerable.TakeWhile Queryable.TakeWhile
Fragmento	Divide los elementos de una secuencia en fragmentos de un tamaño máximo especificado.	No aplicable.	Enumerable.Chunk Queryable.Chunk

Todos los ejemplos siguientes usan [Enumerable.Range\(Int32, Int32\)](#) para generar una secuencia de números de 0 a 7.

Use el método `Take` para tomar solo los primeros elementos de una secuencia:

C#

```
foreach (int number in Enumerable.Range(0, 8).Take(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
```

Use el método `Skip` para omitir los primeros elementos de una secuencia y usar los elementos restantes:

C#

```
foreach (int number in Enumerable.Range(0, 8).Skip(3))
{
    Console.WriteLine(number);
}
```

```
// This code produces the following output:  
// 3  
// 4  
// 5  
// 6  
// 7
```

Los métodos `TakeWhile` y `SkipWhile` también toman y omiten elementos en una secuencia. Sin embargo, en lugar de un número establecido de elementos, estos métodos omiten o toman elementos en función de una condición. `TakeWhile` toma los elementos de una secuencia hasta que un elemento no coincide con la condición.

C#

```
foreach (int number in Enumerable.Range(0, 8).TakeWhile(n => n < 5))  
{  
    Console.WriteLine(number);  
}  
// This code produces the following output:  
// 0  
// 1  
// 2  
// 3  
// 4
```

`SkipWhile` omite los primeros elementos, siempre y cuando la condición sea verdadera. Se devuelve el primer elemento que no coincide con la condición y todos los elementos posteriores.

C#

```
foreach (int number in Enumerable.Range(0, 8).SkipWhile(n => n < 5))  
{  
    Console.WriteLine(number);  
}  
// This code produces the following output:  
// 5  
// 6  
// 7
```

El operador `Chunk` se usa para dividir los elementos de una secuencia en función de un valor `size` determinado.

C#

```
int chunkNumber = 1;  
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))  
{
```

```
Console.WriteLine($"Chunk {chunkNumber++}:");
foreach (int item in chunk)
{
    Console.WriteLine($"      {item}");
}

Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//   0
//   1
//   2
//
//Chunk 2:
//   3
//   4
//   5
//
//Chunk 3:
//   6
//   7
```

El código de C# anterior:

- Se basa en `Enumerable.Range(Int32, Int32)` para generar una secuencia de números.
- Aplica el operador `chunk` y divide la secuencia en fragmentos con un tamaño máximo de tres.

Consulte también

- [System.Linq](#)



Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

[Proporcionar comentarios sobre el producto](#)

Convertir tipos de datos (C#)

Artículo • 31/10/2024

Los métodos de conversión cambian el tipo de los objetos de entrada.

ⓘ Importante

Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan [System.Linq.IQueryable<T>](#) orígenes de datos y [árboles de expresión](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

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.

 Expandir tabla

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
AsEnumerable	Devuelve la entrada con tipo como IEnumerable<T> .	No es aplicable.	Enumerable.AsEnumerable
AsQueryable	Convierte un IEnumerable (genérico) en un IQueryable (genérico).	No es aplicable.	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:	Enumerable.Cast Queryable.Cast
		<pre>from string str in words</pre>	
OfType	Filtrá valores en función de su capacidad para convertirse en un tipo especificado.	No es aplicable.	Enumerable.OfType Queryable.OfType
ToArray	Convierte una colección en una matriz. Este método fuerza la ejecución de la consulta.	No es aplicable.	Enumerable.ToArray
ToDictionary	Coloca elementos en Dictionary< TKey, TValue > basándose en una función de selector de claves. Este método fuerza la ejecución de la consulta.	No es aplicable.	Enumerable.ToDictionary
ToList	Convierte una colección en List< T > . Este método fuerza la ejecución de la consulta.	No es aplicable.	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 aplicable.	Enumerable.ToLookup

⚠️ Nota

En los ejemplos siguientes de este artículo se usan los orígenes de datos comunes para esta área.

Cada `Student` tiene un nivel académico, un departamento principal y una serie de puntuaciones. Un `Teacher` también tiene una propiedad `City` que identifica el campus donde el profesor imparte clases. Un `Department` tiene un nombre y una referencia a un `Teacher` que actúa como jefe del departamento.

Puede encontrar el conjunto de datos de ejemplo en el repositorio [de](#) origen.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

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.

```
C#  
  
IEnumerable people = students;  
  
var query = from Student student in students  
            where student.Year == GradeLevel.ThirdYear  
            select student;  
  
foreach (Student student in query)  
{  
    Console.WriteLine(student.FirstName);  
}
```

La consulta equivalente se puede expresar mediante la sintaxis del método, como se muestra en el ejemplo siguiente:

```
C#  
  
IEnumerable people = students;  
  
var query = people  
    .Cast<Student>()  
    .Where(student => student.Year == GradeLevel.ThirdYear);  
  
foreach (Student student in query)  
{  
    Console.WriteLine(student.FirstName);  
}
```

Consulte también

- [System.Linq](#)
- [from \(cláusula\)](#)

Operaciones Join en LINQ

Artículo • 15/10/2024

Una *join* 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.

ⓘ Importante

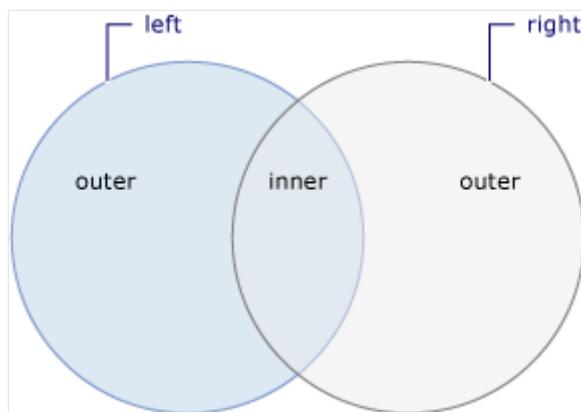
Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan [System.Linq.IQueryable<T>](#) orígenes de datos y [árboles de expresión](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del 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, una combinación 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 `Student` que tiene una propiedad de tipo `Department` que representa su área de especialización, pero la clase `Department` no tiene una propiedad que sea una colección de objetos `Student`. Si tiene una lista de objetos `Department` y quiere encontrar todos los alumnos de cada departamento, podría usar una operación de join para encontrarlos.

Los métodos de join 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. (A modo de comparación, Transact-SQL admite operadores de join distintos de `equals`, por ejemplo, el operador `less than`). En términos de bases de datos relacionales, [Join](#) implementa una join interna, un tipo de join en la que solo se devuelven aquellos objetos que tienen una coincidencia 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 join externa izquierda es una join 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 join interna o en una join externa izquierda.



Métodos

[Expandir tabla](#)

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.	<code>join ... in ... on ... equals ...</code>	Enumerable.Join Queryable.Join
GroupJoin	Combina dos secuencias según las funciones de selector de claves y agrupa los resultados coincidentes para cada elemento.	<code>join ... in ... on ... equals ... into ...</code>	Enumerable.GroupJoin Queryable.GroupJoin

Nota

En los ejemplos siguientes de este artículo se usan los orígenes de datos comunes para esta área.

Cada `Student` tiene un nivel académico, un departamento principal y una serie de puntuaciones. Un `Teacher` también tiene una propiedad `city` que identifica el campus donde el profesor imparte clases. Un `Department` tiene un nombre y una referencia a un `Teacher` que actúa como jefe del departamento.

Puede encontrar el conjunto de datos de ejemplo en el repositorio [de](#)

```

public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}

```

En el ejemplo siguiente se usa la cláusula `join ... in ... on ... equals ...` para join dos secuencias en función de un valor específico:

C#

```

var query = from student in students
            join department in departments on student.DepartmentID equals
department.ID
            select new { Name = $"{student.FirstName} {student.LastName}",

DepartmentName = department.Name };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}

```

La consulta anterior se puede expresar mediante la sintaxis del método, como se muestra en el código siguiente:

```
C#  
  
var query = students.Join(departments,  
    student => student.DepartmentID, department => department.ID,  
    (student, department) => new { Name = $"{student.FirstName}  
{student.LastName}", DepartmentName = department.Name });  
  
foreach (var item in query)  
{  
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");  
}
```

En el ejemplo siguiente se usa la cláusula `join ... in ... on ... equals ... into ...` para join dos secuencias en función de un valor específico y se agrupan las coincidencias resultantes de cada elemento:

```
C#  
  
IEnumerable<IEnumerable<Student>> studentGroups = from department in  
departments  
        join student in students on department.ID equals  
student.DepartmentID into studentGroup  
        select studentGroup;  
  
foreach (IEnumerable<Student> studentGroup in studentGroups)  
{  
    Console.WriteLine("Group");  
    foreach (Student student in studentGroup)  
    {  
        Console.WriteLine($" - {student.FirstName}, {student.LastName}");  
    }  
}
```

La consulta anterior se puede expresar mediante la sintaxis del método, como se muestra en el ejemplo siguiente:

```
C#  
  
// Join department and student based on DepartmentId and grouping result  
IEnumerable<IEnumerable<Student>> studentGroups =  
departments.GroupJoin(students,  
    department => department.ID, student => student.DepartmentID,  
    (department, studentGroup) => studentGroup);  
  
foreach (IEnumerable<Student> studentGroup in studentGroups)  
{
```

```

Console.WriteLine("Group");
foreach (Student student in studentGroup)
{
    Console.WriteLine($" - {student.FirstName}, {student.LastName}");
}

```

Realizar combinaciones internas

En términos de la base de datos relacional, una *join 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 join interna. En los siguientes ejemplos se muestra cómo realizar cuatro variaciones de una join interna:

- Una join interna simple que correlaciona elementos de dos orígenes de datos según una clave simple.
- Una join 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 *join múltiple* en la que las sucesivas operaciones de join se anexan entre sí.
- Una join interna que se implementa mediante una join agrupada.

join de clave única

En el ejemplo siguiente se comparan objetos `Teacher` con objetos `Department` cuyos elementos `TeacherId` coinciden con esos objetos `Teacher`. La cláusula `select` de C# define el aspecto que tendrán los objetos resultantes. En el ejemplo siguiente, los objetos resultantes son tipos anónimos que constan del nombre del departamento y el nombre del profesor que dirige el departamento.

C#

```

var query = from department in departments
            join teacher in teachers on department.TeacherID equals
teacher.ID
            select new
            {
                DepartmentName = department.Name,
                TeacherName = $"{teacher.First} {teacher.Last}"
            };

```

```
foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Se obtienen los mismos resultados mediante la sintaxis del método `Join`:

C#

```
var query = teachers
    .Join(departments, teacher => teacher.ID, department =>
department.TeacherID,
        (teacher, department) =>
        new { DepartmentName = department.Name, TeacherName = $""
{teacher.First} {teacher.Last}" });

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Los profesores que no son jefes de departamento no aparecen en los resultados finales.

join 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. 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 `Teacher` y una lista de objetos `Student` para determinar qué profesores son también alumnos. Ambos tipos tienen propiedades que representan el nombre y el apellido de cada persona. Las funciones que crean las claves de join de los elementos de cada lista devuelven un tipo anónimo formado por las propiedades. La operación de join compara la igualdad de estas claves compuestas y devuelve pares de objetos de cada lista en los que el nombre y el apellido coinciden.

C#

```
// 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 teacher in teachers
    join student in students on new
    {
        FirstName = teacher.First,
        LastName = teacher.Last
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select teacher.First + " " + teacher.Last;

string result = "The following people are both teachers and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.WriteLine(result);

```

Puede usar el método `Join`, tal y como se muestra en el siguiente ejemplo:

C#

```

IEnumerable<string> query = teachers
    .Join(students,
        teacher => new { FirstName = teacher.First, LastName = teacher.Last },
        student => new { student.FirstName, student.LastName },
        (teacher, student) => $"{teacher.First} {teacher.Last}"
    );

Console.WriteLine("The following people are both teachers and students:");
foreach (string name in query)
{
    Console.WriteLine(name);
}

```

join múltiple

Se puede anexar cualquier número de operaciones de join entre sí para realizar una join múltiple. Cada cláusula `join` de C# correlaciona un origen de datos especificado con los resultados de la join anterior.

La primera cláusula `join` correlaciona los alumnos y los departamentos en función de la coincidencia del valor `DepartmentID` de un objeto `student` con el valor `ID` de un objeto

`Department`. Devuelve una secuencia de tipos anónimos que contienen los objetos `Student` y `Department`.

La segunda cláusula `join` correlaciona los tipos anónimos devueltos por la primera `join` con objetos `Teacher` basados en el ID de ese profesor que coinciden con el ID del jefe de departamento. Devuelve una secuencia de tipos anónimos que contienen el nombre del alumno, el nombre del departamento y el nombre del jefe del departamento. Dado que esta operación es una `join` interna, solo se devuelven los objetos del primer origen de datos que tienen una correspondencia en el segundo origen de datos.

C#

```
// The first join matches Department.ID and Student.DepartmentID from the
// list of students and
// departments, based on a common ID. The second join matches teachers who
// lead departments
// with the students studying in that department.
var query = from student in students
    join department in departments on student.DepartmentID equals
department.ID
    join teacher in teachers on department.TeacherID equals teacher.ID
    select new {
        StudentName = $"{student.FirstName} {student.LastName}",
        DepartmentName = department.Name,
        TeacherName = $"{teacher.First} {teacher.Last}"
    };
foreach (var obj in query)
{
    Console.WriteLine($"The student "{obj.StudentName}" studies in the
department run by "{obj.TeacherName}".");
}
```

El equivalente que usa varios métodos `Join` usa el mismo enfoque con el tipo anónimo:

C#

```
var query = students
    .Join(departments, student => student.DepartmentID, department =>
department.ID,
    (student, department) => new { student, department })
    .Join(teachers, commonDepartment =>
commonDepartment.department.TeacherID, teacher => teacher.ID,
    (commonDepartment, teacher) => new
    {
        StudentName = $"{commonDepartment.student.FirstName}
{commonDepartment.student.LastName}",
        DepartmentName = commonDepartment.department.Name,
        TeacherName = $"{teacher.First} {teacher.Last}"
    });
});
```

```

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the
department run by "{obj.TeacherName}"");
}

```

join interna mediante join agrupada

El ejemplo siguiente muestra cómo implementar una join interna mediante una join agrupada. La lista de objetos `Department` forma una combinación agrupada con la lista de objetos `Student` según el `Department.ID` que coincide con la propiedad `Student.DepartmentID`. La join agrupada crea una colección de grupos intermedios, donde cada grupo consta de un objeto `Department` y una secuencia de objetos `Student` coincidentes. La segunda cláusula `from` combina (o acopla) esta secuencia de secuencias en una secuencia más larga. La cláusula `select` especifica el tipo de elementos de la secuencia final. Ese tipo es un tipo anónimo que consta del nombre del alumno y del nombre del departamento coincidente.

C#

```

var query1 =
    from department in departments
    join student in students on department.ID equals student.DepartmentID
    into gj
    from subStudent in gj
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    };
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}

```

Los mismos resultados se pueden lograr mediante el método `GroupJoin`, como se indica a continuación:

C#

```

var queryMethod1 = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, gj) => new { department, gj })
    .SelectMany(departmentAndStudent => departmentAndStudent.gj,

```

```

        (departmentAndStudent, subStudent) => new
    {
        DepartmentName = departmentAndStudent.department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    });
}

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in queryMethod1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}

```

El resultado es equivalente al conjunto de resultados obtenido con la cláusula `join` sin la cláusula `into` para realizar una join interna. En el código siguiente se muestra esta consulta equivalente:

C#

```

var query2 = from department in departments
    join student in students on department.ID equals student.DepartmentID
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    };

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in query2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}

```

Para evitar el encadenamiento, el método `Join` único se puede usar como se muestra aquí:

C#

```

var queryMethod2 = departments.Join(students, departments => departments.ID,
    student => student.DepartmentID,
    (department, student) => new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    });

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in queryMethod2)
{

```

```
        Console.WriteLine($"'{v.DepartmentName} - {v.StudentName}'");
    }
```

Realizar combinaciones agrupadas

La join 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.

ⓘ Nota

Cada elemento de la primera colección aparece en el conjunto de resultados de una join 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 join no agrupada, que no puede acceder a los elementos de la primera colección que no tienen ninguna coincidencia en la segunda colección.

⚠ Advertencia

[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 join agrupada. Para más información, consulte [Entity Framework Core, GroupJoin](#).

En el primer ejemplo de este artículo se muestra cómo realizar una join agrupada. En el segundo ejemplo se muestra cómo usar una join agrupada para crear elementos XML.

join agrupada

En el ejemplo siguiente se realiza una join agrupada de objetos de tipo `Department` y `Student` basada en la coincidencia de `Department.ID` con la propiedad `Student.DepartmentID`. A diferencia de una join no agrupada, que genera un par de elementos para cada coincidencia, la join agrupada produce un único objeto resultante para cada elemento de la primera colección, que en este ejemplo es un objeto `Department`. Los elementos correspondientes de la segunda colección, que en este

ejemplo son objetos `Student`, se agrupan en una colección. Por último, la función de selector de resultados crea un tipo anónimo para cada coincidencia formada por `Department.Name` y una colección de objetos `Student`.

C#

```
var query = from department in departments
            join student in students on department.ID equals student.DepartmentID
            into studentGroup
            select new
            {
                DepartmentName = department.Name,
                Students = studentGroup
            };

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($" {student.FirstName} {student.LastName}");
    }
}
```

En el ejemplo anterior, la variable `query` contiene la consulta que crea una lista donde cada elemento es un tipo anónimo que contiene el nombre del departamento y una colección de alumnos que estudian en ese departamento.

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```
var query = departments.GroupJoin(students, department => department.ID,
                                    student => student.DepartmentID,
                                    (department, Students) => new { DepartmentName = department.Name,
                                    Students });

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($" {student.FirstName} {student.LastName}");
    }
}
```

```
    }  
}
```

join 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.

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",  
    from department in departments  
    join student in students on department.ID equals student.DepartmentID  
    into studentGroup  
    select new XElement("Department",  
        new XAttribute("Name", department.Name),  
        from student in studentGroup  
        select new XElement("Student",  
            new XAttribute("FirstName", student.FirstName),  
            new XAttribute("LastName", student.LastName))  
    )  
);  
  
Console.WriteLine(departmentsAndStudents);
```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",  
    departments.GroupJoin(students, department => department.ID, student =>  
    student.DepartmentID,  
    (department, Students) => new XElement("Department",  
        new XAttribute("Name", department.Name),  
        from student in Students  
        select new XElement("Student",  
            new XAttribute("FirstName", student.FirstName),  
            new XAttribute("LastName", student.LastName))  
    )  
)  
);  
  
Console.WriteLine(departmentsAndStudents);
```

Realizar operaciones de combinación externa izquierda

Una join externa izquierda es una join 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 join externa izquierda llamando al método [DefaultIfEmpty](#) en los resultados de una join agrupada.

En el ejemplo siguiente se muestra cómo usar el método [DefaultIfEmpty](#) en los resultados de una join agrupada para realizar una join externa izquierda.

El primer paso para generar una join externa izquierda de dos colecciones consiste en realizar una join interna usando una join agrupada. (Vea [Realizar combinaciones internas](#) para obtener una explicación de este proceso). En este ejemplo, la lista de objetos `Department` está unida mediante combinación interna a la lista de objetos `Student` basándose en el id. de un objeto `Department` que coincide con el elemento `DepartmentID`.

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 join agrupada. En este ejemplo, se llama a [DefaultIfEmpty](#) en cada secuencia de objetos `Student` coincidentes. El método devuelve una colección que contiene un único, valor predeterminado si la secuencia de objetos `Student` coincidentes está vacía para cualquier objeto `Department`, con lo que cada objeto `Department` se representa en la colección de resultados.

ⓘ Nota

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 `Student`.

C#

```
var query =
    from student in students
    join department in departments on student.DepartmentID equals
    department.ID into gj
    from subgroup in gj.DefaultIfEmpty()
    select new
    {
```

```

        student.FirstName,
        student.LastName,
        Department = subgroup?.Name ?? string.Empty
    };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
    }
}

```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```

var query = students.GroupJoin(departments, student => student.DepartmentID,
department => department.ID,
    (student, departmentList) => new { student, subgroup =
departmentList.AsQueryable() })
    .SelectMany(joinedSet => joinedSet.subgroup.DefaultIfEmpty(), (student,
department) => new
{
    student.student.FirstName,
    student.student.LastName,
    Department = department.Name
});

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}

```

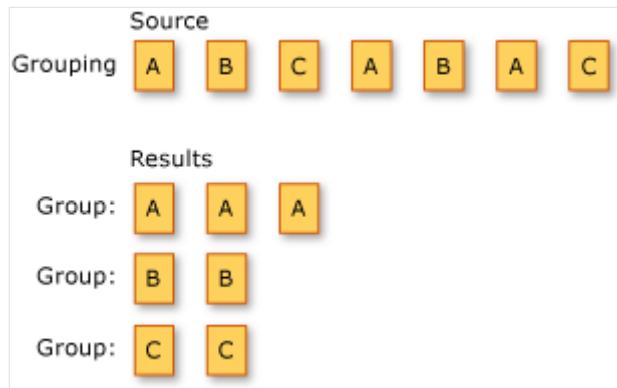
Consulte también

- [Join](#)
- [GroupJoin](#)
- [Tipos anónimos \(Guía de programación de C#\)](#).
- [Cómo: Formular combinaciones y consultas entre productos](#)
- [join clause](#)
- [group \(cláusula\)](#)
- [Procedimiento para join contenido de archivos no similares \(LINQ\) \(C#\)](#)
- [Procedimiento para llenar colecciones de objetos de varios orígenes \(LINQ\) \(C#\)](#)

Agrupar datos (C#)

Artículo • 04/06/2024

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.



ⓘ Importante

Estos ejemplos usan un origen de datos

[System.Collections.Generic.IEnumerable<T>](#). Los orígenes de datos basados en [System.Linq.IQueryProvider](#) usan [System.Linq.IQueryable<T>](#) orígenes de datos y [árboles de expresión](#). Los árboles de expresión tienen [limitaciones](#) en la sintaxis de C# permitida. Además, cada origen de datos [IQueryProvider](#), como [EF Core](#) puede imponer más restricciones. Compruebe la documentación del origen de datos.

Los métodos de operador de consulta estándar que agrupan elementos de datos se enumeran en la tabla siguiente.

 Expandir tabla

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
GroupBy	Agrupa los elementos que comparten un atributo común. Un objeto IGrouping< TKey, TElement > representa cada grupo.	<code>group ... by</code> o bien	Enumerable.GroupBy Queryable.GroupBy

Nombre del método	Descripción	Sintaxis de la expresión de consulta de C#	Más información
		group ... by ... into ...	
ToLookup	Inserta elementos a una <code>Lookup< TKey, TElement ></code> (un diccionario uno a varios) basándose en una función de selector de claves.	No aplicable.	Enumerable.ToLookup

El ejemplo de código siguiente usa la cláusula `group by` para agrupar los enteros de una lista según sean pares o impares.

C#

```
List<int> numbers = [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);
    }
}
```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = numbers
    .GroupBy(number => number % 2);

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

```
    }  
}
```

ⓘ Nota

En los ejemplos siguientes de este artículo se usan los orígenes de datos comunes para esta área.

Cada `Student` tiene un nivel académico, un departamento principal y una serie de puntuaciones. Un `Teacher` también tiene una propiedad `City` que identifica el campus donde el profesor imparte clases. Un `Department` tiene un nombre y una referencia a un `Teacher` que actúa como jefe del departamento.

Puede encontrar el conjunto de datos de ejemplo en el repositorio [de ↗ origen](#).

C#

```
public enum GradeLevel  
{  
    FirstYear = 1,  
    SecondYear,  
    ThirdYear,  
    FourthYear  
};  
  
public class Student  
{  
    public required string FirstName { get; init; }  
    public required string LastName { get; init; }  
    public required int ID { get; init; }  
  
    public required GradeLevel Year { get; init; }  
    public required List<int> Scores { get; init; }  
  
    public required int DepartmentID { get; init; }  
}  
  
public class Teacher  
{  
    public required string First { get; init; }  
    public required string Last { get; init; }  
    public required int ID { get; init; }  
    public required string City { get; init; }  
}  
  
public class Department  
{  
    public required string Name { get; init; }  
    public int ID { get; init; }
```

```
    public required int TeacherID { get; init; }  
}
```

Agrupar los resultados de consultas

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 alumno. Para obtener más información, vea la [cláusula group](#).

Ejemplo de agrupación por propiedad única

En el ejemplo siguiente se muestra cómo agrupar elementos de origen mediante una propiedad única del elemento como la clave de grupo. La clave es un elemento `enum`, el año del alumno en la escuela. La operación de agrupación usa al comparador de igualdad predeterminado para el tipo.

C#

```
var groupByYearQuery =  
    from student in students  
    group student by student.Year into newGroup  
    orderby newGroup.Key  
    select newGroup;  
  
foreach (var yearGroup in groupByYearQuery)  
{  
    Console.WriteLine($"Key: {yearGroup.Key}");  
    foreach (var student in yearGroup)  
    {  
        Console.WriteLine($"{student.LastName}, {student.FirstName}");  
    }  
}
```

El código equivalente mediante la sintaxis del método se muestra en el ejemplo siguiente:

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByYearQuery = students
    .GroupBy(student => student.Year)
    .OrderBy(newGroup => newGroup.Key);

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}
```

Ejemplo de agrupación por valor

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.

C#

```
var groupByFirstLetterQuery =
    from student in students
    let firstLetter = student.LastName[0]
    group student by firstLetter;

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}
```

Se requiere foreach anidado para acceder a los elementos de grupo.

El código equivalente mediante la sintaxis del método se muestra en el ejemplo siguiente:

C#

```
var groupByFirstLetterQuery = students
    .GroupBy(student => student.LastName[0]);
```

```

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}

```

Ejemplo de agrupación por intervalo

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.

C#

```

static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
    orderby percentGroup.Key
    select percentGroup;

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"{item.LastName}, {item.FirstName}");
    }
}

```

Se requiere foreach anidado para iterar en grupos y elementos de grupo. El código equivalente mediante la sintaxis del método se muestra en el ejemplo siguiente:

```
C#  
  
static int GetPercentile(Student s)  
{  
    double avg = s.Scores.Average();  
    return avg > 0 ? (int)avg / 10 : 0;  
}  
  
var groupByPercentileQuery = students  
    .Select(student => new { student, percentile = GetPercentile(student) })  
    .GroupBy(student => student.percentile)  
    .Select(percentGroup => new  
    {  
        percentGroup.Key,  
        Students = percentGroup.Select(s => new { s.student.FirstName,  
s.student.LastName })  
    })  
    .OrderBy(percentGroup => percentGroup.Key);  
  
foreach (var studentGroup in groupByPercentileQuery)  
{  
    Console.WriteLine($"Key: {studentGroup.Key * 10}");  
    foreach (var item in studentGroup.Students)  
    {  
        Console.WriteLine($"{item.LastName}, {item.FirstName}");  
    }  
}
```

Ejemplo de agrupación por comparación

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. Las propiedades del tipo anónimo se convierten en propiedades en el miembro `Key`.

```
C#  
  
var groupByHighAverageQuery =  
    from student in students  
    group new  
    {  
        student.FirstName,  
        student.LastName  
    } by student.Scores.Average() > 75 into studentGroup
```

```

    select studentGroup;

    foreach (var studentGroup in groupByHighAverageQuery)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"{student.FirstName} {student.LastName}");
        }
    }
}

```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```

var groupByHighAverageQuery = students
    .GroupBy(student => student.Scores.Average() > 75)
    .Select(group => new
    {
        group.Key,
        Students = group.AsEnumerable().Select(s => new { s.FirstName,
s.LastName })
    });

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup.Students)
    {
        Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}

```

Agrupación por tipo anónimo

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.

C#

```

var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetterOfLastName = student.LastName[0],
        ScoreAbove85 = student.Scores.Average() > 85
    }
    into studentGroup
    select studentGroup;

```

```

        IsScoreOver85 = student.Scores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetterOfLastName
    select studentGroup;

    foreach (var scoreGroup in groupByCompoundKey)
    {
        var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
        Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"{item.FirstName} {item.LastName}");
        }
    }
}

```

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```

var groupByCompoundKey = students
    .GroupBy(student => new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    })
    .OrderBy(studentGroup => studentGroup.Key.FirstLetterOfLastName);

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"{item.FirstName} {item.LastName}");
    }
}

```

Crear un grupo anidado

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.

C#

```

var nestedGroupsQuery =
    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;

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
    {
        Console.WriteLine($"    Names that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"        {innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}

```

Se necesitan tres bucles `foreach` anidados para recorrer en iteración los elementos internos de un grupo anidado.

(Mantenga el cursor del mouse sobre las variables de iteración, `outerGroup`, `innerGroup` y `innerGroupElement` para ver su tipo real).

La consulta equivalente mediante la sintaxis del método se muestra en el código siguiente:

C#

```

var nestedGroupsQuery =
    students
    .GroupBy(student => student.Year)
    .Select(newGroup1 => new
    {
        newGroup1.Key,
        NestedGroup = newGroup1
            .GroupBy(student => student.LastName)
    });

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup.NestedGroup)
    {
        Console.WriteLine($"    Names that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {

```

```
        Console.WriteLine($"\\t\\t{innerGroupElement.LastName}\\n{innerGroupElement.FirstName}");
    }
}
```

Realizar una subconsulta en una operación de agrupación

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. 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.

C#

```
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.Scores.Average()
        ).Max()
    };
var 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.

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Max(student2 =>
student2.Scores.Average())
        });

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
}
```

Consulte también

- [System.Linq](#)
- [GroupBy](#)
- [IGrouping<TKey,TElement>](#)
- [group \(cláusula\)](#)
- [Procedimiento para dividir un archivo en varios mediante el uso de grupos \(LINQ\) \(C#\)](#)

Procedimientos para usar LINQ para consultar archivos y directorios

Artículo • 02/05/2024

Muchas operaciones de sistema de archivos son esencialmente consultas y, por tanto, son adecuadas para el enfoque LINQ. Estas consultas no son destructivas. No cambian el contenido de las carpetas o los archivos originales. Las consultas no deberían provocar efectos secundarios. En general, cualquier código (incluidas las consultas que ejecutan operaciones de creación actualización y eliminación) que modifica los datos de origen se debe separar del código que solo consulta los datos.

Hay cierta complejidad en la creación de un origen de datos que represente de forma precisa el contenido del sistema de archivos y controle 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 las instancias de [FileInfo](#) se 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 acceder a la propiedad `Length` en una consulta, el objeto [FileInfo](#) intentará acceder 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.

Procedimiento para buscar archivos con un nombre o atributo especificados

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. Es posible que tenga que modificar la primera línea de muchos de los ejemplos, independientemente de que este código se ejecute en Windows, Mac o en un sistema Linux.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
```

```

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;

// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//     Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();

Console.WriteLine($"\\r\\nThe newest .txt file is {newestFile.FullName}.
Creation time: {newestFile.CreationTime}");

```

Procedimiento para agrupar archivos por extensión

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](#).

En la siguiente consulta se muestra cómo agrupar el contenido de un árbol de directorio especificado por la extensión de nombre de archivo.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fileGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

```

```

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine($"{f.FullName.Substring(trimLength)}");
    }
    Console.WriteLine();
}

```

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 requiere un bucle `foreach` anidado, ya que cada grupo se enumera por separado.

Procedimiento para consultar el número total de bytes en un conjunto de carpetas

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. El método `Sum` agrega los valores de todos los elementos seleccionados en la cláusula `select`. Puede modificar esta consulta para recuperar el archivo más grande, o el más pequeño, del árbol del directorio especificado llamando al método `Min` o `Max`, en lugar de `Sum`.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
    let fileLen = new FileInfo(file).Length
    where fileLen > 0
    select fileLen;

// 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($"There are {totalBytes} bytes in {fileList.Count()} files
under {startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

En este ejemplo se amplía el ejemplo anterior para hacer lo siguiente:

- 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.

El ejemplo siguiente contiene cinco consultas independientes que muestran cómo consultar y agrupar archivos, en función de su tamaño en bytes. Puede modificar estos ejemplos para basar la consulta en otra propiedad del objeto [FileInfo](#).

C#

```

        let len = fileInfo.Length
        orderby len descending
        select fileInfo
    ).Take(10);

Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups = from file in fileList
    let fileInfo = new FileInfo(file)
    let len = fileInfo.Length
    where len > 0
    group fileInfo by (len / 100000) into fileGroup
    where fileGroup.Key >= 2
    orderby fileGroup.Key descending
    select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}0000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"{item.Name}: {item.Length}");
    }
}

```

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.

Procedimiento para consultar archivos duplicados en un árbol de directorios

A veces, archivos que tienen el mismo nombre pueden estar en más de una carpeta. 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.

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.",
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;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {({queryDup.Key.ToString() == string.Empty
? "[none]" : queryDup.Key.ToString()})}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"{fileName}");
    }
}

```

La primera consulta usa una clave para determinar una coincidencia. Encuentra archivos que tienen 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.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Make the lines shorter for the console display
int charsToSkip = startFolder.Length;

// Take a snapshot of the file system.
DirectoryInfo dir = new DirectoryInfo(startFolder);
IEnumerable<FileInfo> fileList = dir.GetFiles("*.",
SearchOption.AllDirectories);

// Note the use of a compound key. Files that match

```



```

        let fileText = File.ReadAllText(file.FullName)
        where fileText.Contains(searchTerm)
        select file.FullName;

// Execute the query.
Console.WriteLine($"""The term \"{searchTerm}\" was found in:""");
foreach (string filename in queryMatchingFiles)
{
    Console.WriteLine(filename);
}

```

Procedimiento para: comparar el contenido de dos carpetas

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.

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. No está previsto que la clase se use 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.

C#

```

// 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 : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, 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(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}

public static void CompareDirectories()
{
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles(".*",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles(".*",
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 (total
number = {queryCommonFiles.Count()});");
        foreach (var v in queryCommonFiles.Take(10))
    }
}

```

```

        {
            Console.WriteLine(v.Name); //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.
var queryList1Only = (from file in list1
                      select file)
                      .Except(list2, myFileCompare);

Console.WriteLine();
Console.WriteLine($"The following files are in list1 but not list2
(total number = {queryList1Only.Count()});");
foreach (var v in queryList1Only.Take(10))
{
    Console.WriteLine(v.FullName);
}

var queryList2Only = (from file in list2
                      select file)
                      .Except(list1, myFileCompare);

Console.WriteLine();
Console.WriteLine($"The following files are in list2 but not list1
(total number = {queryList2Only.Count()});");
foreach (var v in queryList2Only.Take(10))
{
    Console.WriteLine(v.FullName);
}
}

```

Procedimiento para reordenar los campos de un archivo delimitado

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. Si se usa el método [Split](#) para separar los campos, es muy fácil consultar y manipular archivos CSV mediante LINQ. De hecho, se puede usar la misma técnica para reordenar los elementos de cualquier línea estructurada de texto, no solo los archivos CSV.

En el siguiente ejemplo, suponga que las tres columnas representan el "apellido", el "nombre" y el "identificador" de los alumnos. Los campos están ordenados

alfabéticamente por el apellido de los alumnos. La consulta genera una nueva secuencia en la que la columna del identificador aparece en primer lugar, seguida de 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. En el texto siguiente se muestra el contenido del archivo *spreadsheet1.csv* usado en el ejemplo siguiente:

txt

```
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
```

El siguiente código lee el archivo de origen y reorganiza todas las columnas del archivo CSV para cambiar su orden:

C#

```
string[] lines = 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 fields = line.Split(',')
    orderby fields[2]
    select $"{fields[2]}, {fields[1]} {fields[0]}";

File.WriteAllLines("spreadsheet2.csv", query.ToArray());

/* 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
```

Procedimiento para dividir un archivo en varios mediante grupos

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. La consulta usa el contenido de dos archivos. En el texto siguiente se muestra el contenido del primer archivo, *names1.txt*:

txt

Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra

El segundo archivo, *names2.txt*, contiene un conjunto diferente de nombres, algunos de los cuales son comunes con el primer conjunto:

txt

Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi

El código siguiente consulta ambos archivos, toma su unión y, después, escribe un archivo para cada grupo, definido por la primera letra del apellido:

C#

```
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
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(',')[0]
                  group name by n[0] into g
                  orderby g.Key
                  select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);

    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
        Console.WriteLine($"    {item}");
    }
}
/* 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
```

Procedimiento para combinar contenido de archivos no similares

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.

En el siguiente texto se muestra el contenido de *scores.csv*. 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.

txt

```
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
```

En el siguiente texto se muestra el contenido de *names.csv*. El archivo representa una hoja de cálculo que contiene el nombre, el primer apellido y el identificador de los estudiantes.

txt

```
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

Combine contenido de archivos diferentes que contienen información relacionada. El archivo *names.csv* contiene el nombre del alumno y un número de identificación. El archivo *scores.csv* contiene el identificador y un conjunto de cuatro puntuaciones de prueba. La siguiente consulta combina las puntuaciones con los nombres de los alumnos y usa el identificador como clave de coincidencia. El código se muestra en el siguiente ejemplo:

C#

```
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = 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]}";

Console.WriteLine("\r\nMerge two spreadsheets:");
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
Console.WriteLine("{0} total names in list", scoreQuery.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
*/
```

Procedimiento para calcular valores de columna en un archivo de texto CSV

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.

En el siguiente texto se muestra el contenido de `scores.csv`. Imagínese que la primera columna representa un identificador de estudiante y que las columnas siguientes representan las notas de cuatro exámenes.

```
txt  
  
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
```

En el siguiente texto se muestra cómo usar 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.

```
C#  
  
public class SumColumns  
{  
    public static void SumCSVColumns(string fileName)  
    {  
        string[] lines = File.ReadAllLines(fileName);  
  
        // 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);  
    }  
}
```



```

        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++)
    {
        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 #{column + 1} Average: {average:##.##}
High Score: {max} Low Score: {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
*/

```

Si el archivo es un archivo separado por tabulaciones, actualice el argumento del método `Split` a `\t`.



Colaborar con nosotros en



Comentarios de .NET

GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento: Uso de LINQ para consultar cadenas

Artículo • 21/05/2024

Las cadenas se almacenan como una secuencia de caracteres. Como secuencia de caracteres, se pueden consultar mediante LINQ. En este artículo, hay varias consultas de ejemplo que consultan distintos caracteres o palabras en las cadenas, filtran cadenas o mezclan consultas con expresiones regulares.

Procedimiento para consultar caracteres en una cadena

En el ejemplo siguiente se consulta una cadena para determinar el número de dígitos numéricos que contiene.

C#

```
string aString = "ABCDE99F-J74-12-89A";

// Select only those characters that are numbers
var 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 = {count}");

// Select all characters before the first '-'
var stringQuery2 = aString.TakeWhile(c => c != '-');

// Execute the second query
foreach (char c in stringQuery2)
    Console.Write(c);
/* Output:
   Output: 9 9 7 4 1 2 8 9
   Count = 8
   ABCDE99F
*/
```

En la consulta anterior se muestra cómo se puede tratar una cadena como una secuencia de caracteres.

Procedimiento para: realizar el recuento de las repeticiones de una palabra en una cadena

En el ejemplo siguiente se muestra cómo usar una consulta LINQ para contar las repeticiones de una palabra determinada en una cadena. Para realizar el recuento, primero se llama al método [Split](#) para crear una matriz de palabras. El método [Split](#) implica un costo de rendimiento. Si la única operación de la cadena es para contar las palabras, debe considerar la posibilidad de usar los métodos [Matches](#) o [IndexOf](#) en su lugar.

```
C#  
  
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 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  
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];  
string[] source = text.Split(separators,  
StringSplitOptions.RemoveEmptyEntries);  
  
// Create the query. Use the InvariantCultureIgnoreCase comparison to match  
"data" and "Data"  
var matchQuery = from word in source  
    where word.Equals(searchTerm,  
 StringComparison.InvariantCultureIgnoreCase)  
    select word;  
  
// Count the matches, which executes the query.  
int wordCount = matchQuery.Count();
```

```
Console.WriteLine($"""{wordCount} occurrences(s) of the search term "
{searchTerm}" were found."");
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

En la consulta anterior se muestra cómo puede ver cadenas como una secuencia de palabras, después de dividir una cadena en una secuencia de palabras.

Procedimiento para ordenar o filtrar datos de texto por cualquier palabra o campo

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 de forma dinámica en tiempo de ejecución. Imagine que los campos de scores.csv representan el número de identificación de un alumno, seguido de una serie de cuatro calificaciones:

txt

```
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
```

En la consulta siguiente se ordenan las líneas en función de la puntuación del primer examen, almacenada en la segunda columna:

C#

```
// Create an IEnumerable data source
string[] scores = File.ReadAllLines("scores.csv");

// Change this to any value from 0 to 4.
int sortField = 1;

Console.WriteLine($"Sorted highest to lowest by field [{sortField}]:");

// Split the string and sort on field[num]
```

```

var scoreQuery = from line in scores
    let fields = line.Split(',')
    orderby fields[sortField] descending
    select line;

foreach (string str in scoreQuery)
{
    Console.WriteLine(str);
}
/* 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 la consulta anterior se muestra cómo puede manipular cadenas si las divide en campos y consulta los campos individuales.

Consulta de oraciones con palabras específicas

En el ejemplo siguiente 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, también se podría llenar de forma dinámica en tiempo de ejecución. La consulta devuelve las frases que contienen las palabras "Historically", "data" e "integrated".

C#

```

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 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(['.', '?', '!']);
```

```

// Define the search terms. This list could also be dynamically populated at
run time.
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.
char[] separators = [ '.', '?', '!', ' ', ';', ':' ];
var sentenceQuery = from sentence in sentences
    let w =
        sentence.Split(separators, StringSplitOptions.RemoveEmptyEntries)
            where w.Distinct().Intersect(wordsToMatch).Count() ==
wordsToMatch.Count()
    select sentence;
```

```

foreach (string str in sentenceQuery)
{
    Console.WriteLine(str);
}
/* 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 cada frase 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.

La llamada a `Split` usa signos de puntuación como separadores para quitarlos de la cadena. Si no quitara los signos de puntuación, por ejemplo, podría tener una cadena "Historically," que no coincidiría con "Historically" en la matriz `wordsToMatch`. Es posible que tenga que usar separadores adicionales, en función de los tipos de puntuación del texto de origen.

Procedimiento para: Combinar consultas LINQ con expresiones regulares

En el ejemplo siguiente se muestra cómo usar la clase [Regex](#) a fin de 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.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Take a snapshot of the file system.
var fileList = from file in Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories)
    let fileInfo = new FileInfo(file)
    select fileInfo;

// Create the regular expression to find all things "Visual".
System.Text.RegularExpressions.Regex searchTerm =
    new System.Text.RegularExpressions.Regex(@"microsoft.net.
(sdk|workload)");

// 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 == ".txt"
    let fileText = 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 \"{searchTerm}\" was found in:""");

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}");
}
```

También puede consultar el objeto [MatchCollection](#) devuelto por una búsqueda [RegEx](#). En los resultados solo se muestra el valor de cada coincidencia. Pero también es posible usar LINQ para realizar todo tipo de operaciones de filtrado, ordenación y agrupación en esa colección. Como [MatchCollection](#) no es una colección [IEnumerable](#) genérica, tendrá que indicar explícitamente el tipo de la variable de rango en la consulta.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

LINQ y colecciones

Artículo • 28/04/2024

La mayoría de las colecciones modelan una *secuencia* de elementos. Puede usar LINQ para consultar cualquier tipo de colección. Otros métodos LINQ buscan elementos de una colección, valores de proceso de los elementos de una colección o modifican la colección o sus elementos. Estos ejemplos le ayudan a obtener información sobre los métodos LINQ y cómo puede usarlos con las colecciones u otros orígenes de datos.

Procedimiento para buscar la diferencia de conjuntos entre dos listas

En este ejemplo se muestra cómo usar LINQ para comparar dos listas de cadenas y generar estas líneas, que se encuentran en la primera colección, pero no en la segunda. La primera colección de nombres se almacena en el archivo *names1.txt*:

```
txt  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Potra, Cristina  
Noriega, Fabricio  
Aw, Kam Foo  
Beebe, Ann  
Toyoshima, Tim  
Guy, Wey Yuan  
Garcia, Debra
```

La segunda colección de nombres se almacena en el archivo *names2.txt*. Algunos nombres aparecen en ambas secuencias.

```
txt  
Liu, Jinghao  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Beebe, Ann  
Gilchrist, Beth  
Myrcha, Jacek  
Giakoumakis, Leo  
McLin, Nkenge  
El Yassir, Mehdi
```

En el código siguiente se muestra cómo puede usar el método [Enumerable.Except](#) para buscar elementos en la primera lista que no están en la segunda:

```
C#  
  
// Create the IEnumerable data sources.  
string[] names1 = File.ReadAllLines("names1.txt");  
string[] names2 = File.ReadAllLines("names2.txt");  
  
// Create the query. Note that method syntax must be used here.  
var 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);  
/* 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, como [Except](#), [Distinct](#), [Union](#) y [Concat](#), solo pueden expresarse en una sintaxis basada en métodos.

Procedimiento para: combinar y comparar colecciones de cadenas

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, una unión y una intersección en los dos conjuntos de líneas de texto. Use los mismos dos archivos de texto que se muestran en el ejemplo anterior. El código muestra ejemplos de [Enumerable.Concat](#), [Enumerable.Union](#) y [Enumerable.Except](#).

```
C#  
  
//Put text files in your solution folder  
string[] fileA = File.ReadAllLines("names1.txt");  
string[] fileB = File.ReadAllLines("names2.txt");  
  
//Simple concatenation and sort. Duplicates are preserved.  
var 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.
var 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).
var 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";

var tempQuery1 = from name in fileA
                let n = name.Split(',')
                where n[0] == nameMatch
                select name;

var tempQuery2 = from name2 in fileB
                let n2 = name2.Split(',')
                where n2[0] == nameMatch
                select name2;

var nameMatchQuery = tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"""Concat based on partial name match
{nameMatch}":""");

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine($"{query.Count()} total names in list");
}
/* 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, Nkeng  
Myrcha, Jacek  
Noriega, Fabricio  
Potra, Cristina  
Toyoshima, Tim  
20 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, Nkeng  
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
```

```
*/
```

Procedimiento para llenar colecciones de objetos de varios orígenes

En este ejemplo se muestra cómo combinar datos de orígenes diferentes en una secuencia de tipos nuevos.

⚠ Nota

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.

En este ejemplo se usan dos archivos. El primero, `names.csv`, contiene nombres de alumno e identificadores de alumno.

txt

```
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
```

El segundo, `scores.csv`, contiene identificadores de alumno en la primera columna, seguidos de las puntuaciones del examen.

txt

```
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
```

En el ejemplo siguiente se muestra cómo usar un registro `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. El identificador se usa como clave para asignar alumnos a sus puntuaciones.

C#

```
// 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 = 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 = 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(',')
== Convert.ToInt32(splitScoreLine[0])
                                             where Convert.ToInt32(splitName[2])
                                             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)
                                             ).ToArray()
);

// 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 {student.FirstName}
```

```

{student.LastName} is {student.ExamScores.Average()});  

}  

/* 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`, cada nuevo objeto `Student` se inicializa a partir de los datos de los dos orígenes.

Si no tiene que almacenar los resultados de una consulta, las tuplas o los tipos anónimos pueden ser más convenientes que los tipos con nombre. En el ejemplo siguiente se ejecuta la misma tarea que en el ejemplo anterior, con la diferencia de que se usan tuplas en lugar de tipos con nombre:

C#

```

// 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 (FirstName: splitName[0],  

                                LastName: 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 {student.FirstName}

```

```
{student.LastName} is {student.ExamScores.Average()}."};  
}
```

Procedimiento para consultar un objeto ArrayList con LINQ

Cuando use LINQ para consultar colecciones no genéricas `IEnumerable`, como `ArrayList`, deberá declarar explícitamente el tipo de variable de rango para reflejar el tipo específico de los objetos de la colección. Por ejemplo, si tiene un elemento `ArrayList` de `Student` objetos, la cláusula `from` deberá tener un aspecto similar a este:

C#

```
var query = from Student s in arrList  
//...
```

Al especificar el tipo de la variable de rango, se convierte cada elemento del objeto `ArrayList` en un objeto `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](#). En el siguiente ejemplo se muestra una consulta sobre un objeto [ArrayList](#).

C#

```
ArrayList arrList = new ArrayList();
arrList.Add(
    new Student
    (
        FirstName: "Svetlana",
        LastName: "Omelchenko",
        ExamScores: new int[] { 98, 92, 81, 60 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Claire",
        LastName: "O'Donnell",
        ExamScores: new int[] { 75, 84, 91, 39 }
    ));
arrList.Add(
    new Student
    (
```

```
        FirstName: "Sven",
        LastName: "Mortensen",
        ExamScores: new int[] { 88, 94, 65, 91 }
    );
arrList.Add(
    new Student
    (
        FirstName: "Cesar",
        LastName: "Garcia",
        ExamScores: new int[] { 97, 89, 85, 82 }
    ));

var query = from Student student in arrList
            where student.ExamScores[0] > 95
            select student;

foreach (Student s in query)
    Console.WriteLine(s.LastName + ":" + s.ExamScores[0]);
```

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Cómo extender LINQ

Artículo • 26/04/2024

Todos los métodos basados en LINQ siguen uno de los dos patrones similares. Toman una secuencia enumerable. Devuelven una secuencia diferente o un único valor. La coherencia de la forma permite extender LINQ escribiendo métodos con una forma similar. De hecho, las bibliotecas de .NET obtuvieron nuevos métodos en muchas versiones de .NET desde que se introdujo LINQ por primera vez. En este artículo, verá ejemplos de extensión de LINQ escribiendo sus propios métodos siguiendo el mismo patrón.

Agregar métodos personalizados para las consultas LINQ

Para extender el conjunto de métodos que usa para consultas LINQ, agregue 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 solo 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 nueva secuencia. 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 obtener más información, vea [Métodos de extensión](#).

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`.

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
```

```

    if (source is null || !source.Any())
    {
        throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
    }

    var sortedList =
        source.OrderBy(number => number).ToList();

    int itemIndex = sortedList.Count / 2;

    if (sortedList.Count % 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`.

C#

```

double[] numbers = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//      double: Median = 4.85

```

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.

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 `int`.

C#

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

Ahora puede llamar a las sobrecargas `Median` para los tipos `integer` y `double`, como se muestra en el código siguiente:

C#

```
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
//     int: Median = 3
```

También puede crear una sobrecarga que acepte una *secuencia genérica* de objetos. 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`.

C#

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (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.

C#

```
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:
//      int: Median = 3
//      string: Median = 4
```

Extienda la interfaz `IEnumerable<T>` con un método de consulta personalizado que devuelva 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.

C#

```
// 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)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
```

```

        yield return element;
    }

    index++;
}
}

```

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:

C#

```

string[] strings = ["a", "b", "c", "d", "e"];

var query5 = stringsAlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}

// This code produces the following output:
//      a
//      c
//      e

```

Agrupar resultados por claves contiguas

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:

[] Expandir tabla

Key	Value
A	We
A	think
A	that
B	Linq
C	is
A	really

Key	Value
B	cool
B	!

Los siguientes grupos se crean en este orden:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

La solución se implementa como método de extensión seguro para subprocessos que devuelve los resultados mediante transmisión por secuencias. Genera sus grupos a medida que se desplaza por la secuencia de origen. A diferencia de los operadores `group` o `orderby`, es posible empezar a devolver grupos al autor de la llamada antes de leer toda la secuencia. En el ejemplo siguiente se muestran el método de extensión y el código de cliente que lo usa:

C#

```
public static class ChunkExtensions
{
    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
    TKey>(
        this IEnumerable<TSource> source,
        Func<TSource, TKey> keySelector) =>
        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.
        IEnumerator<TSource>? enumerator = source.GetEnumerator();

        // Move to the first element in the source sequence.
        if (!enumerator.MoveNext())
        {
            yield break;           // source collection is empty
        }

        while (true)
```

```

    {
        var key = keySelector(enumerator.Current);

        Chunk<TKey, TSource> current = new(key, enumerator, value =>
comparer.Equals(key, keySelector(value))));

        yield return current;

        if (current.CopyAllChunkElements() == noMoreSourceElements)
        {
            yield break;
        }
    }
}

```

C#

```

public static class GroupByContiguousKeys
{
    // The source sequence.
    static readonly KeyValuePair<string, string>[] list = [
        new("A", "We"),
        new("A", "think"),
        new("A", "that"),
        new("B", "LINQ"),
        new("C", "is"),
        new("A", "really"),
        new("B", "cool"),
        new("B", "!")
    ];

    // Query variable declared as class member to be available
    // on different threads.
    static readonly IEnumerable<IGrouping<string, KeyValuePair<string,
string>>> query =
        list.ChunkBy(p => p.Key);

    public static void GroupByContiguousKeys1()
    {
        // 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($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"{inner.Value}");
            }
        }
    }
}

```

Clase ChunkExtensions

En el código presentado de la implementación de la clase `ChunkExtensions`, el bucle `while(true)` del método `ChunkBy` recorre en iteración la secuencia de origen y crea una copia de cada fragmento. En cada paso, el iterador avanza hasta el primer elemento del siguiente "Fragmento", representado por un objeto `Chunk` en la secuencia de origen. Este bucle corresponde al bucle `foreach` externo que ejecuta la consulta. En ese bucle, el código realiza las siguientes acciones:

1. Obtenga la clave del fragmento actual y asígnela a la variable `key`. El iterador de origen consume la secuencia de origen hasta que encuentre un elemento con una clave que no coincida.
2. Cree un nuevo objeto de Fragmento (grupo) y almacénelo en la variable `current`. Tiene un objeto `GroupItem`, una copia del elemento de origen actual.
3. Devuelve ese fragmento. Un fragmento es un `IGrouping<TKey, TSource>`, que es el valor devuelto del método `ChunkBy`. El Fragmento solo tiene el primer elemento de su secuencia de origen. Los elementos restantes solo se devolverán cuando el `foreach` del código de cliente se aplique sobre este fragmento. Consulte `Chunk.GetEnumerator` para obtener más información.
4. Compruebe si:
 - El fragmento tuviera una copia de todos sus elementos de origen o
 - El iterador alcanzó el final de la secuencia de origen.
5. Cuando el autor de la llamada enumeró todos los elementos del fragmento, el método `Chunk.GetEnumerator` habrá copiado todos los elementos del fragmento. Si el bucle `Chunk.GetEnumerator` no enumerase todos los elementos del fragmento, hágalo en este momento para evitar dañar el iterador para los clientes que pudieran llamarlo en un subproceso independiente.

Clase Chunk

La clase `Chunk` es un grupo contiguo de uno o varios elementos de origen que tienen la misma clave. Un fragmento tiene una clave y una lista de objetos `ChunkItem`, que son copias de los elementos de la secuencia de origen:

C#

```
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 Value;
    public ChunkItem? Next;
}

public TKey Key { get; }

// 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;

// Private object for thread synchronization
private readonly object m_Lock;

// REQUIRES: enumerator != null && predicate != null
public Chunk(TKey key, [DisallowNull] Ienumerator<TSource> enumerator,
[DisallowNull] Func<TSource, bool> predicate)
{
    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.
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.
    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.
    if (isLastSourceElement || !predicate(enumerator.Current))
    {
        enumerator = default!;
        predicate = default!;
    }
    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.
internal bool CopyAllChunkElements()
{
    while (true)
    {
        lock (m_Lock)
        {
            if (DoneCopyingChunk)
            {
                return isLastSourceElement;
            }
            else
            {
                CopyNextChunkElement();
            }
        }
    }
}

// Stays just one step ahead of the client requests.
public IEnumrator<TSource> GetEnumrator()
{
    // 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();
}

```

Cada `ChunkItem` (representado por `ChunkItem` clase) tiene una referencia a la siguiente `ChunkItem` de la lista. La lista consta de su `head`, que almacena el contenido del primer elemento de origen que pertenece a este fragmento y es `tail`, que es el final de la lista. Se cambia la posición de la cola cada vez que se agrega un nuevo `ChunkItem`. La cola de la lista de vínculo se establece en `null` en el método `CopyNextChunkElement` si la clave del elemento siguiente no coincide con la clave del fragmento actual o no hay más elementos en el origen.

El método `CopyNextChunkElement` de la clase `Chunk` agrega un `ChunkItem` al grupo actual de elementos. Intenta avanzar el iterador en la secuencia de origen. Si el método `MoveNext()` devolviese `false`, la iteración estará al final y `isLastSourceElement` se establece en `true`.

Se llama al método `CopyAllChunkElements` después de que se alcance el final del último fragmento. Comprueba si hay más elementos en la secuencia de origen. Si los hubiera, devolverá `true` si se agotó el enumerador de este fragmento. En este método, cuando se comprueba el campo privado `DoneCopyingChunk` es revisado para `true`, si `isLastSourceElement` es `false`, indicará al iterador externo para continuar iterando.

El bucle `foreach` interno invoca el método `GetEnumerator` de la clase `Chunk`. Este método permanece un elemento por delante de las solicitudes de cliente. Agrega el siguiente elemento del fragmento solo después de que el cliente solicite el último elemento anterior de la lista.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Consulta basada en el estado en tiempo de ejecución

Artículo • 03/05/2024

En la mayoría de las consultas LINQ, la forma general de la consulta se establece en el código. Puede filtrar elementos mediante una cláusula `where`, ordenar la colección de salida mediante `orderby`, elementos de grupo o realizar algún cálculo. El código puede proporcionar parámetros para el filtro, la clave de ordenación u otras expresiones que forman parte de la consulta. Sin embargo, la forma general de la consulta no puede cambiar. En este artículo, aprenderá técnicas para usar la interfaz `System.Linq.IQueryable<T>` y los tipos que lo implementan para modificar la forma de una consulta en tiempo de ejecución.

Estas técnicas se usan para crear consultas en tiempo de ejecución, donde algunos datos de entrada de usuario o estado en tiempo de ejecución cambian los métodos de consulta que desea usar como parte de la consulta. Quiere editar la consulta al agregar, quitar o modificar cláusulas de consulta.

ⓘ Nota

Asegúrese de agregar `using System.Linq.Expressions;` y `using static System.Linq.Expressions.Expression;` en la parte superior del archivo de .cs.

Tenga en cuenta el código que define una interfaz `IQueryable` o `IQueryable<T>` con respecto a un origen de datos:

C#

```
string[] companyNames = [
    "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"
];

// Use an in-memory array as the data source, but the IQueryable could have
// come
// from anywhere -- an ORM backed by a database, a web request, or any other
// LINQ provider.
```

```
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Cada vez que ejecute el código anterior, se ejecuta la misma consulta exacta. Vamos a aprender a modificar la consulta para ampliarla o modificarla. Fundamentalmente, una interfaz [IQueryable](#) tiene dos componentes:

- [Expression](#): representación independiente del lenguaje y del origen de datos de los componentes de la consulta actual, en forma de un árbol de expresión.
- [Provider](#), una instancia de un proveedor LINQ, que sabe cómo materializar la consulta actual en un valor o en un conjunto de valores.

En el contexto de las consultas dinámicas, el proveedor normalmente sigue siendo el mismo; el árbol de expresión de la consulta varía entre consultas.

Los árboles de expresión son inmutables; si quiere otro árbol de expresión (y, por tanto, otra consulta), tendrá que convertir el existente en uno nuevo. En las secciones siguientes, se describen técnicas específicas para realizar consultas de forma diferente en respuesta al estado del entorno de ejecución:

- Uso del estado del entorno de ejecución desde el árbol de expresión
- Llamar a más métodos LINQ
- Variación del árbol de expresión que se pasa a los métodos de LINQ
- Creación de un árbol de expresión [Expression<TDelegate>](#) con los métodos de generador de [Expression](#)
- Adición de nodos de llamada de método al árbol de expresión de [IQueryable](#)
- Construcción de cadenas y uso de la [biblioteca dinámica de LINQ](#) ↗

Cada una de las técnicas permite más funcionalidades, pero a un costo de mayor complejidad.

Uso del estado del entorno de ejecución desde el árbol de expresión

La manera más sencilla de realizar consultas dinámicas consiste en hacer referencia al estado del entorno de ejecución de forma directa en la consulta mediante una variable cerrada, como `length` en el ejemplo de código siguiente:

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
```

```
.Distinct();

Console.WriteLine(string.Join(", ", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(", ", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

El árbol de expresión interno (y, por tanto, la consulta) no se han modificado; la consulta solo devuelve otros valores porque se ha cambiado el valor de `length`.

Llamar a más métodos LINQ

Por lo general, los [métodos de LINQ integrados](#) en `Queryable` realizan dos pasos:

- Encapsulan el árbol de expresión actual en un elemento `MethodCallExpression` que representa la llamada de método.
- Vuelven a pasar el árbol de expresión encapsulado al proveedor, ya sea para devolver un valor mediante el método `IQueryProvider.Execute` del proveedor, o bien para devolver un objeto de consulta traducido mediante el método `IQueryProvider.CreateQuery`.

Puede reemplazar la consulta original con el resultado de un método que devuelva `System.Linq.IQueryable<T>` para obtener una nueva consulta. Puede usar el estado en tiempo de ejecución, como en el ejemplo siguiente:

```
C#  
  
// bool sortByLength = /* ... */;  
  
var qry = companyNamesSource;  
if (sortByLength)  
{  
    qry = qry.OrderBy(x => x.Length);  
}
```

Variación del árbol de expresión que se pasa a los métodos de LINQ

Puede pasar otras expresiones a los métodos de LINQ, en función del estado del entorno de ejecución:

```
C#
```

```

// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("", "") => x => true,
    (_, "") => x => x.StartsWith(startsWith),
    ("", _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);

```

También es posible que quiera crear las distintas subexpresiones con otra biblioteca, como [PredicateBuilder](#) de [LinqKit](#):

C#

```

// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);

```

Creación de árboles de expresión y consultas mediante métodos de generador

En todos los ejemplos vistos hasta ahora, se conoce el tipo de elemento en tiempo de compilación (`string`) y, por tanto, el tipo de la consulta (`IQueryable<string>`). Es posible que agregue componentes a una consulta de cualquier tipo de elemento o agregue

componentes diferentes, en función del tipo de elemento. Puede crear árboles de expresión desde cero, con los métodos de generador de [System.Linq.Expressions.Expression](#) y, por tanto, adaptar la expresión en tiempo de ejecución a un tipo de elemento específico.

Creación de una expresión<TDelegate>

Cuando se crea una expresión para pasarl a uno de los métodos de LINQ, en realidad se crea una instancia de [System.Linq.Expressions.Expression<TDelegate>](#), donde `TDelegate` es un tipo de delegado como `Func<string, bool>`, `Action` o un tipo de delegado personalizado.

[System.Linq.Expressions.Expression<TDelegate>](#) hereda de [LambdaExpression](#), que representa una expresión lambda completa como el siguiente ejemplo:

C#

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

[LambdaExpression](#) tiene dos componentes:

1. Una lista de parámetros (`(string x)`) representada por la propiedad [Parameters](#).
2. Un cuerpo (`x.StartsWith("a")`) representado por la propiedad [Body](#).

Los pasos básicos para crear una instancia de [Expression<TDelegate>](#) son los siguientes:

1. Defina objetos [ParameterExpression](#) para cada uno de los parámetros (si existen) de la expresión lambda, mediante el método generador [Parameter](#).

C#

```
ParameterExpression x = Parameter(typeof(string), "x");
```

2. Construya el cuerpo de [LambdaExpression](#) utilizando los valores [ParameterExpression](#) definidos y los métodos Factory Method en [Expression](#). Por ejemplo, una expresión que represente `x.StartsWith("a")` se podría crear de la siguiente manera:

C#

```
Expression body = Call(  
    x,  
    typeof(string).GetMethod("StartsWith", [typeof(string)])!,
```

```
    Constant("a")
);
```

3. Ajuste los parámetros y el cuerpo en una instancia de `Expression<TDelegate>` con tipo de tiempo de compilación, mediante la sobrecarga apropiada de Factory Method [Lambda](#):

```
C#  
  
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,  
x);
```

En las secciones siguientes, se describe un escenario en el que es posible que desee construir una `Expression<TDelegate>` para pasarla a un método LINQ. Proporciona un ejemplo completo de cómo hacerlo mediante los métodos de fábrica.

Construcción de una consulta completa en tiempo de ejecución

Quiere escribir consultas que funcionen con varios tipos de entidad:

```
C#  
  
record Person(string LastName, string FirstName, DateTime DateOfBirth);  
record Car(string Model, int Year);
```

En cualquiera de estos tipos de entidad, quiere filtrar y devolver solo las entidades que contengan un texto concreto dentro de uno de sus campos `string`. Para `Person`, le interesa buscar las propiedades `FirstName` y `LastName`:

```
C#  
  
string term = /* ... */;  
var personsQry = new List<Person>()  
    .AsQueryable()  
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Pero para `Car`, solo quiere buscar la propiedad `Model`:

```
C#  
  
string term = /* ... */;  
var carsQry = new List<Car>()
```

```
.AsQueryable()
    .Where(x => x.Model.Contains(term));
```

Aunque podría escribir una función personalizada para `IQueryable<Person>` y otra para `IQueryable<Car>`, la siguiente función agrega este filtrado a cualquier consulta existente, con independencia del tipo de elemento específico.

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties = elementType
        .GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains",
    [typeof(string)])!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
            // node like x.PropertyName.Contains("term")
            Call(
                Property(prm, prp), // .Contains(...)
                Constant(term) // "term"
            )
        );
    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate((prev, current) => Or(prev, current));
}
```

```

// Wrap the expression body in a compile-time-typed lambda expression
Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

// Because the lambda is compile-time-typed (albeit with a generic
// parameter), we can use it with the Where method
return source.Where(lambda);
}

```

Como la función `TextFilter` toma y devuelve una interfaz `IQueryable<T>` (y no solo una interfaz `IQueryable`), puede agregar más elementos de consulta con tipo de tiempo de compilación después del filtro de texto.

C#

```

var qry = TextFilter(
    new List<Person>().AsQueryable(),
    "abcd"
)
.Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
    new List<Car>().AsQueryable(),
    "abcd"
)
.Where(x => x.Year == 2010);

```

Agregar nodos de llamada de método al árbol de expresiones de `IQueryable<TDelegate>`

Si tiene una interfaz `IQueryable` en lugar de `IQueryable<T>`, no puede llamar directamente a los métodos de LINQ genéricos. Una alternativa consiste en crear el árbol de expresión interno como se ha indicado en el ejemplo anterior y usar la reflexión para invocar el método de LINQ adecuado mientras se pasa el árbol de expresión.

También puede duplicar la función del método de LINQ y encapsular todo el árbol en un elemento `MethodCallExpression` que represente una llamada al método de LINQ:

C#

```

IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
    // LambdaExpression's body is the same as in the previous example,

```

```

// but has been refactored into the constructBody function.
(Expression? body, ParameterExpression? prm) =
constructBody(elementType, term);
if (body is null) { return source; }

Expression filteredTree = Call(
    typeof(Queryable),
    "Where",
    [elementType],
    source.Expression,
    Lambda(body, prm!)
);

return source.Provider.CreateQuery(filteredTree);
}

```

En este caso no tiene un marcador de posición genérico `T` en tiempo de compilación, por lo que usará la sobrecarga de `Lambda` que no necesita información de tipos de tiempo de compilación y que genera un elemento `LambdaExpression` en lugar de una `Expression<TDelegate>`.

Biblioteca dinámica de LINQ

La creación de árboles de expresión mediante métodos de generador es relativamente compleja; es más fácil crear cadenas. La [biblioteca dinámica de LINQ](#) expone un conjunto de métodos de extensión en `IQueryable` correspondiente a los métodos estándar de LINQ en `Queryable` y que aceptan cadenas en una [sintaxis especial](#) en lugar de árboles de expresión. La biblioteca genera el árbol de expresión adecuado a partir de la cadena y puede devolver la interfaz `IQueryable` traducida resultante.

El ejemplo anterior se podría volver a escribir de esta manera:

C#

```

// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    var query = source
        .Select("new(" + stringProperties[0].Name + ")");
    if (term != null)
        query = query.Where(stringProperties[0].Name + " == " + term);
    return query;
}

```

```
// Build the string expression
string filterExpr = string.Join(
    " || ",
    stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
);

return source.Where(filterExpr, term);
}
```

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Asynchronous programming with `async` and `await`

Article • 09/28/2023

The [Task asynchronous programming model \(TAP\)](#) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a [Task](#) that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks are complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first task completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each

cook (or thread) would be blocked synchronously waiting for the bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
    // example. They are simply marker classes for the purpose of demonstration,
    // contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    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");

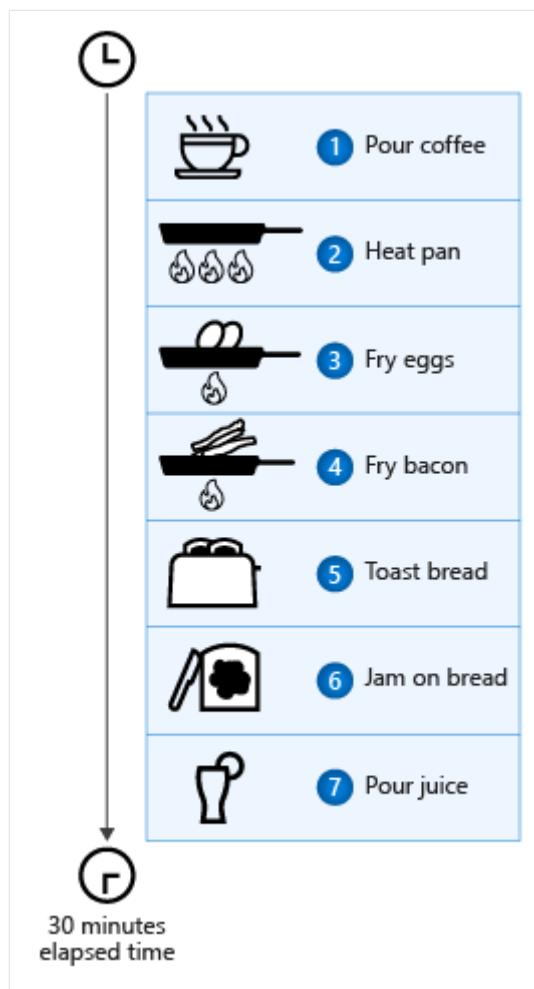
    return new Bacon();
}

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();
}
}

```



The synchronously prepared breakfast took roughly 30 minutes because the total is the sum of each task.

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had been completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that

its step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

```
C#  
  
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

The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

💡 Tip

The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the [final version](#) later in this article.

⚠ Note

The `Main` method returns `Task`, despite not having a `return` expression—this is by design. For more information, see [Evaluation of a void-returning `async` function](#).

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The `System.Threading.Tasks.Task` and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then wait for something else that requires your attention.

You start a task and hold on to the `Task` object that represents the work. You'll `await` each task before working with its result.

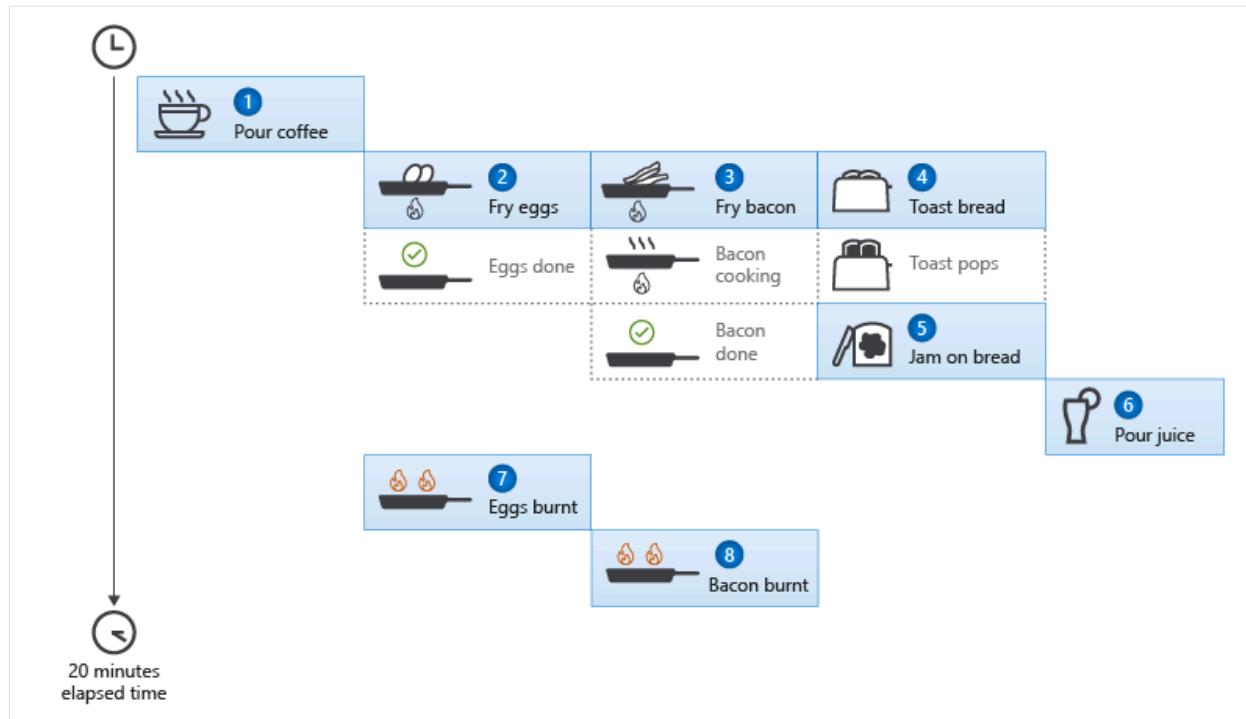
Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```
C#  
  
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!");
```

The preceding code won't get your breakfast ready any faster. The tasks are all awaited as soon as they are started. Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

```
C#  
  
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!");
```



The asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests to different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

ⓘ Important

The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use `Task` or `Task<TResult>` objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a `Task<TResult>` that represents the composition of those three operations. The main block of code now becomes:

C#

```
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!");
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is awaited. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

C#

```
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(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

ⓘ Note

You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

You'll notice quite a few tasks are completed between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is *faulted*. The Task object holds the exception thrown in the [Task.Exception](#) property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the [Task](#). The [Task.Exception](#) property is a [System.AggregateException](#) because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the [AggregateException.InnerExceptions](#) collection. If that [Exception](#) property is null, a new [AggregateException](#) is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the [Exception](#) property contains exactly one exception. When code `awaits` a faulted task, the first exception in the [AggregateException.InnerExceptions](#) collection is rethrown. That's why the output from

this example shows an `InvalidOperationException` instead of an `AggregateException`. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario may generate multiple exceptions.

Tip

We recommend that any argument validation exceptions emerge *synchronously* from task-returning methods. For more information and an example of how to do it, see [Exceptions in task-returning methods](#).

Before going on, comment out these two lines in your `ToastBreadAsync` method. You don't want to start another fire:

C#

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is `WhenAll`, which returns a `Task` that completes when all the tasks in its argument list have completed, as shown in the following code:

C#

```
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!");
```

Another option is to use `WhenAny`, which returns a `Task<Task>` that completes when any of its arguments complete. You can await the returned task, knowing that it has already finished. The following code shows how you could use `WhenAny` to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

C#

```
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");
    }
    await finishedTask;
    breakfastTasks.Remove(finishedTask);
}
```

Near the end, you see the line `await finishedTask;`. The line `await Task.WhenAny` doesn't await the finished task. It `awaits` the `Task` returned by `Task.WhenAny`. The result of `Task.WhenAny` is the task that has completed (or faulted). You should `await` that task again, even though you know it's finished running. That's how you retrieve its result, or ensure that the exception causing it to fault gets thrown.

After all those changes, the final version of the code looks like this:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
    // example. They are simply marker classes for the purpose of demonstration,
    // contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    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");
        }
        await finishedTask;
        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();
}

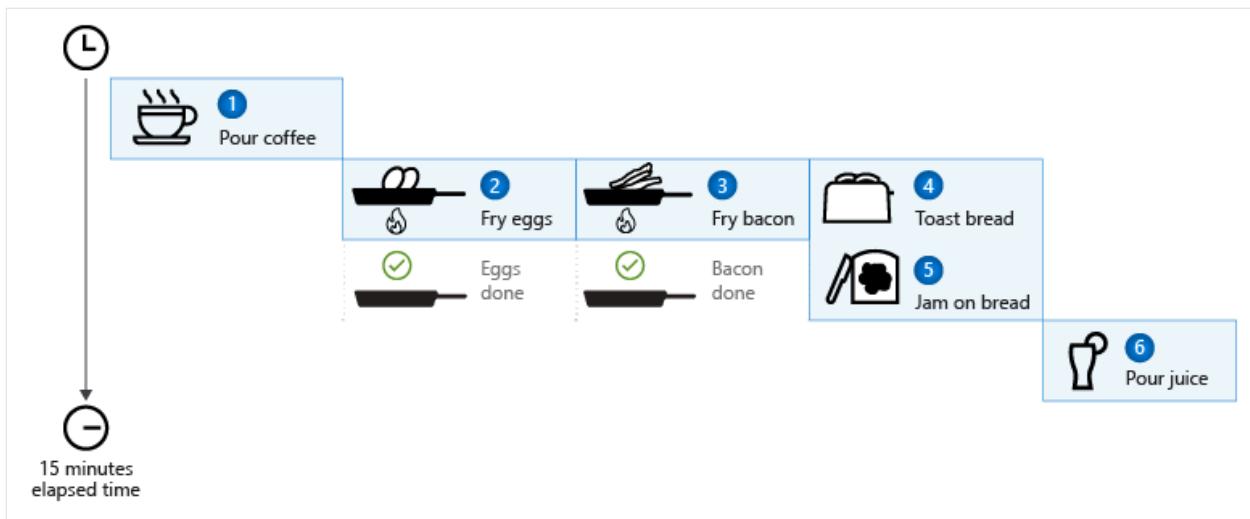
private static async Task<Bacon> FryBaconAsync(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the
pan");
    Console.WriteLine("cooking first side of bacon...");
    await Task.Delay(3000);
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping 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();
}
}
```



The final version of the asynchronously prepared breakfast took roughly 6 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only took action when it was needed.

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Next steps

[Explore real world scenarios for asynchronous programs](#)

Programación asincrónica

Artículo • 10/05/2023

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 admite 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. Esto se puede lograr con la clase [System.Net.Http.HttpClient](#):

C#

```

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 asíncrona) 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.

C#

```

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 subprocesso en segundo plano y lo hace en un modo sin bloqueo.

Qué sucede en segundo plano

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 podría 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.

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.

ⓘ Nota

Si tiene previsto realizar un análisis HTML en el código de producción, no use expresiones regulares. Use una biblioteca de análisis en su lugar.

C#

```
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:

C#

```

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 OnSeeTheDotNetsButtonClick(), 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 asíncrono 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`.

C#

```
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 sencilla, con LINQ:

C#

```
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)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

Aunque es menos código, tenga cuidado al combinar LINQ con código asíncrono. Dado que LINQ usa la ejecución diferida, las llamadas asíncronas 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()`. En el ejemplo anterior se usa `Enumerable.ToArray` para realizar la consulta diligentemente y almacenar los resultados en una matriz. Esto obliga al código `id => GetUserAsync(id)` a ejecutar e iniciar la tarea.

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.

- Agregue "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 subprocesso actual como un medio para esperar que se complete `Task` puede dar lugar a interbloqueos y subprocessos 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...
<code>await</code>	<code>Task.Wait</code> o <code>Task.Result</code>	Recuperar el resultado de una tarea en segundo plano
<code>await</code> <code>Task.WhenAny</code>	<code>Task.WaitAny</code>	Esperar que finalice cualquier tarea
<code>await</code> <code>Task.WhenAll</code>	<code>Task.WaitAll</code>	Esperar que finalicen todas las tareas
<code>await Task.Delay</code>	<code>Thread.Sleep</code>	Esperar un período de tiempo

- **Considere la posibilidad de usar `ValueTask` siempre que 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 usar `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 awaite. 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 predecible, que se puede probar y es fácil de mantener.

Otros recursos

- [Modelo de programación asincrónica de tareas \(C#\).](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Modelo de programación asincrónica de tareas

Artículo • 28/03/2023

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# admite el enfoque simplificado, la programación asincrónica, que aprovecha la compatibilidad asincrónica en el entorno de ejecución de .NET. 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	Windows.Web.Http.HttpClient SyndicationClient
Trabajar con archivos	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile

Área de aplicación	Tipos de .NET con métodos asincrónicos	Tipos de Windows Runtime con métodos asincrónicos
Trabajar con imágenes		MediaCapture BitmapEncoder BitmapDecoder
Programar WCF	Operaciones sincrónicas y asincrónicas	

La asincronía es especialmente valiosa para aquellas aplicaciones que obtienen acceso al subprocesso de interfaz de usuario, ya que todas las actividades relacionadas con la interfaz de usuario normalmente comparten un único subprocesso. 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 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 ejemplo completo de Windows Presentation Foundation (WPF) disponible para su descarga en el artículo [Programación asincrónica con `async` y `await` en C#](#).

```

public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}

```

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` (`contents`). 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 `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` no puede continuar hasta que se complete `getStringTask`.
- Mientras tanto, el control vuelve al autor de la llamada de `GetUrlContentLengthAsync`.
- 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 `GetUrlContentLengthAsync` recuperan el valor de longitud.

Si `GetUrlContentLengthAsync` 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 contents = await  
client.GetStringAsync("https://learn.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`.

Para obtener más información, consulte la sección [Tipos de valor devuelto y parámetros](#).

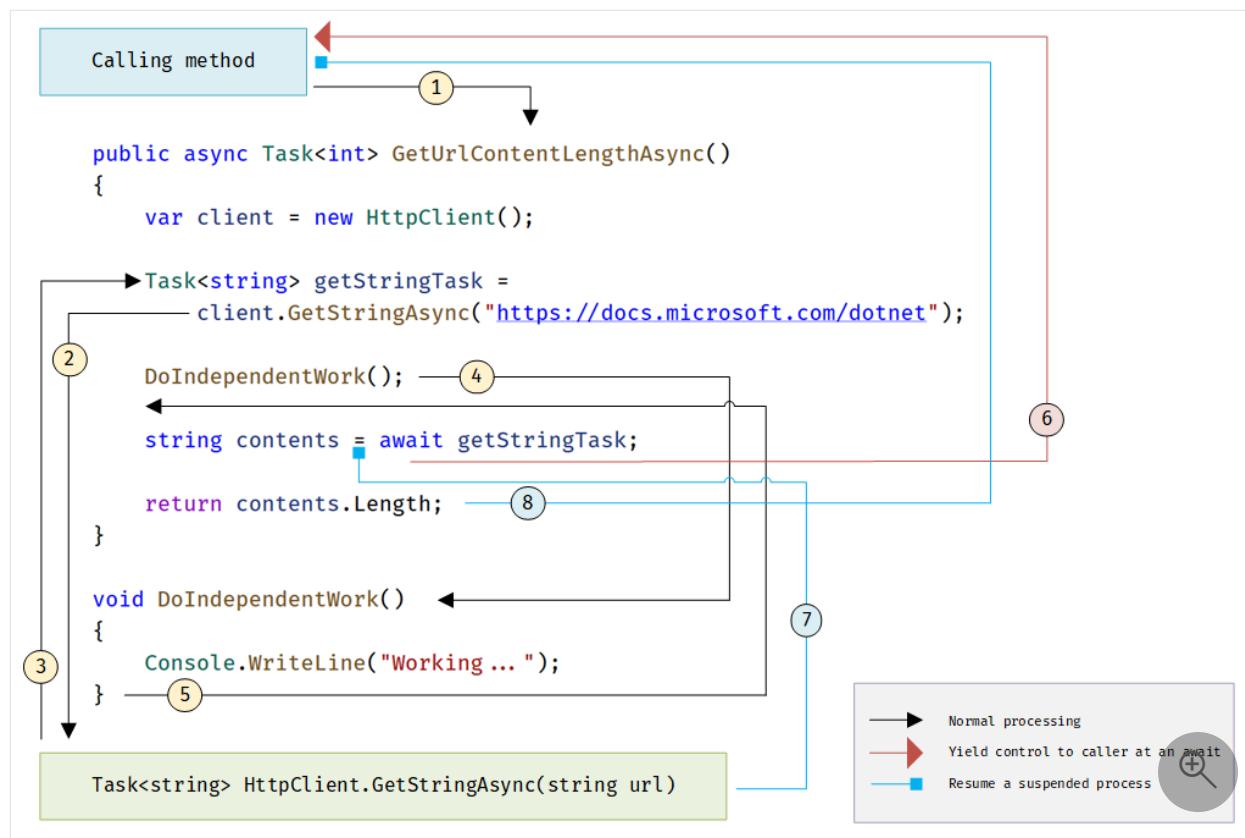
- El método normalmente incluye al menos una expresión `await`, que marca un punto en el que el método no puede continuar hasta que se completa la operación asincrónica en espera. Mientras tanto, se suspende el método y el control vuelve al llamador del método. La sección siguiente de este tema muestra 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 asíncrona 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 con los pasos siguientes, que se inician cuando un método de llamada llama al método asíncrono.

1. Un método de llamada llama al método asíncrono `GetUrlContentLengthAsync` y lo espera.
2. `GetUrlContentLengthAsync` crea una instancia `HttpClient` y llama al método asíncrono `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, `GetUrlContentLengthAsync`.
`GetStringAsync` devuelve un elemento `Task<TResult>`, donde `TResult` es una cadena y `GetUrlContentLengthAsync` 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, `GetUrlContentLengthAsync` 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. `GetUrlContentLengthAsync` se ha quedado sin el trabajo que se puede realizar sin un resultado de `getStringTask`. Después, `GetUrlContentLengthAsync` 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, `GetUrlContentLengthAsync` utiliza un operador `await` para suspender el progreso y producir el control al método que llamó `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` devuelve `Task<int>` al llamador. La tarea representa una sugerencia para generar un resultado entero que es la longitud de la cadena descargada.

⚠ Nota

Si `GetStringAsync` (y, por tanto, `getStringTask`) se completa antes de que `GetUrlContentLengthAsync` lo espere, el control permanece en `GetUrlContentLengthAsync`. El gasto de suspensión y de regresar a `GetUrlContentLengthAsync` se desperdiciaría si el proceso asincrónico `getStringTask` al que se llama ya se ha completado y `GetUrlContentLengthAsync` no tiene que esperar el resultado final.

Dentro del método de llamada, el patrón de procesamiento continúa. El llamador puede hacer otro trabajo que no depende del resultado de `GetUrlContentLengthAsync` antes de esperar ese resultado, o es posible que el llamador se espere inmediatamente. El método de llamada espera a `GetUrlContentLengthAsync`, y `GetUrlContentLengthAsync` espera a `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 `contents`.
8. Cuando `GetUrlContentLengthAsync` tiene el resultado de la cadena, el método puede calcular la longitud de la cadena. El trabajo de `GetUrlContentLengthAsync` 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.

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 subprocessos.

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 utilizar `Task` como tipo de valor devuelto si el método no tiene instrucción `return` o tiene una instrucción `return` que no devuelve un operando.

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`:

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}

Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
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 obtener más información y ejemplos, vea [Tipos de valor devueltos asincrónicos \(C#\)](#).

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 `OnButtonClick`.

Artículos relacionados (Visual Studio)

Título	Descripción
Procedimiento para realizar varias solicitudes web en paralelo con <code>async</code> y <code>await</code> (C#)	Demuestra cómo comenzar varias tareas al mismo tiempo.
Tipos de valor devueltos asincrónicos (C#)	Muestra los tipos que los métodos asincrónicos pueden devolver y explica cuándo es apropiado cada uno de ellos.
Cancelación de tareas con un token de cancelación como mecanismo de señalización.	Muestra cómo agregar la siguiente funcionalidad a la solución asincrónica: <ul style="list-style-type: none"> - Cancelación de una lista de tareas (C#) - Cancelación de tareas asincrónicas tras un período de tiempo (C#) - Iniciar varias tareas asincrónicas y procesarlas a medida que se completan (C#)
Usar <code>Async</code> en acceso a archivos (C#)	Enumera y demuestra los beneficios de usar <code>async</code> y <code>await</code> para obtener acceso a archivos.
Modelo asincrónico basado en tareas (TAP)	Describe un modelo asincrónico, el patrón se basa en los tipos <code>Task</code> y <code>Task<TResult></code> .
Vídeos de Async en Channel 9	Proporciona vínculos a una serie de videos sobre programación asincrónica.

Vea también

- [Programación asincrónica con `async` y `await`](#)
- [`async`](#)
- [`await`](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tipos de valor devueltos asincrónicos (C#)

Artículo • 10/05/2023

Los métodos asincrónicos pueden tener los siguientes tipos de valor devuelto:

- [Task](#), para un método asincrónico que realiza una operación pero no devuelve ningún valor.
- [Task<TResult>](#), para un método asincrónico que devuelve un valor.
- [void](#), para un controlador de eventos.
- Cualquier tipo que tenga un método [GetAwaiter](#) accesible. El objeto devuelto por el método [GetAwaiter](#) debe implementar la interfaz [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#).
- [IAsyncEnumerable<T>](#), para un método asincrónico que devuelve una *secuencia asincrónica*.

Para obtener más información sobre los métodos asincrónicos, vea [Programación asincrónica con async y await \(C#\)](#).

También existen varios tipos que son específicos de las cargas de trabajo de Windows:

- [DispatcherOperation](#): para las operaciones asincrónicas limitadas a Windows.
- [IAsyncAction](#), para las acciones asincrónicas en UWP que no devuelven un valor.
- [IAsyncActionWithProgress<TProgress>](#), para las acciones asincrónicas en UWP que notifican el progreso, pero no devuelven un valor.
- [IAsyncOperation<TResult>](#): para las operaciones asincrónicas en UWP que devuelven un valor.
- [IAsyncOperationWithProgress<TResult,TProgress>](#): para las operaciones asincrónicas en UWP que informan del progreso y devuelven un valor.

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 `WaitAndApologizeAsync` no contiene una instrucción `return`, de manera que el método devuelve un objeto `Task`. La devolución de `Task` permite que se espere a `WaitAndApologizeAsync`. El tipo `Task` no incluye una propiedad `Result` porque no tiene ningún valor devuelto.

C#

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    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 WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//     Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

Se espera a `WaitAndApologizeAsync` 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. Cuando el operando derecho de `await` es `Task<TResult>`, la expresión `await` genera un resultado de `T`. Cuando el operando derecho de `await` es `Task`, `await` y su operando son una instrucción.

Puede separar la llamada a `WaitAndApologizeAsync` 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 `WaitAndApologizeAsync` de la espera de la tarea que el método devuelve.

C#

```

Task waitAndApologizeTask = WaitAndApologizeAsync();

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 waitAndApologizeTask;
Console.WriteLine(output);

```

Tipo de valor devuelto Task<TResult>

El tipo de valor devuelto `Task<TResult>` se usa para un método asincrónico que contiene una instrucción `return` en la que el operando es `TResult`.

En el ejemplo siguiente, el método `GetLeisureHoursAsync` contiene una instrucción `return` que devuelve un entero. 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 propiedad `DayOfWeek`.

C#

```

public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//   Today is Wednesday, May 24, 2017
//   Today's hours of leisure: 5

```

Cuando se llama a `GetLeisureHoursAsync` 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 `GetLeisureHoursAsync` de la aplicación de `await`, como se muestra en el código siguiente. Una llamada al método `GetLeisureHoursAsync` que no se espera inmediatamente devuelve `Task<int>`, como se podría esperar de la declaración del método. La tarea se asigna a la variable `getLeisureHoursTask` en el ejemplo. Dado que `getLeisureHoursTask` es `Task<TResult>`, contiene una propiedad `Result` de tipo `TResult`. En este caso, `TResult` representa un tipo entero. Cuando `await` se aplica a `getLeisureHoursTask`, la expresión `await` se evalúa en el contenido de la propiedad `Result` de `getLeisureHoursTask`. El valor se asigna a la variable `ret`.

ⓘ Importante

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 `Main` pueda imprimir el mensaje (`message`) en la consola antes de que finalice la aplicación.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

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 devuelva `void` no puede detectar las excepciones que inicia el método. Es probable que estas excepciones no controladas provoquen un error en 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. Asegúrese de que cualquier método asincrónico que puede iniciar una excepción tiene un tipo de valor devuelto de `Task` o `Task<TResult>`, y que se esperan las llamadas al método.

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.

C#

```
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 readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();
```

```

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

private static void OnButtonClicked1(object? sender, EventArgs e)
{
    Console.WriteLine("    Handler 1 is starting...");
    Task.Delay(100).Wait();
    Console.WriteLine("    Handler 1 is done.");
}

private static async void OnButtonClicked2Async(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.");
    s_tcs.SetResult(true);
}

private static void OnButtonClicked3(object? sender, EventArgs e)
{
    Console.WriteLine("    Handler 3 is starting...");
    Task.Delay(100).Wait();
    Console.WriteLine("    Handler 3 is done.");
}

// Example output:
//
// Before button.Click() is called...
// 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.
// After button.Click() is called...
//     Handler 2 is done.

```

Tipos de valor devueltos asincrónicos generalizados y ValueTask<TResult>

Un método asincrónico puede devolver cualquier tipo que tenga un método `GetAwaiter` accesible que devuelva una instancia de un *tipo awaite*. Además, el tipo devuelto por el método `GetAwaiter` debe tener el atributo

`System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. Puede obtener más información en el artículo sobre [Atributos leídos por el compilador](#) o la especificación de C# para el [Patrón del compilador de tipos de tareas](#).

Esta característica es el complemento de las [expresiones con await](#), que describe los requisitos del operando de `await`. Los tipos de valor devueltos asincrónicos generalizados permiten al compilador generar métodos `async` que devuelven tipos diferentes. Los tipos de valor devueltos asincrónicos generalizados permitían mejoras de rendimiento en las bibliotecas de .NET. 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. En el ejemplo siguiente se usa la estructura `ValueTask<TResult>` para recuperar el valor de dos tiradas de dado.

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }
}
```

```
static async ValueTask<int> RollAsync()
{
    await Task.Delay(500);

    int diceRoll = s_rnd.Next(1, 7);
    return diceRoll;
}
// Example output:
//   Shaking dice...
//   You rolled 8
```

La escritura de un tipo de valor devuelto asincrónico generalizado es un escenario avanzado y está destinado para su uso en entornos especializados. En su lugar, considere la posibilidad de usar los tipos `Task`, `Task<T>` y `ValueTask<T>`, que abarcan la mayoría de los escenarios del código asincrónico.

En C# 10 y versiones posteriores, se puede aplicar el atributo `AsyncMethodBuilder` a un método asincrónico (en lugar de la declaración de tipo de valor devuelto asincrónico) para invalidar el generador de ese tipo. Normalmente, este atributo se aplica para usar un generador diferente proporcionado en el entorno de ejecución de .NET.

Secuencias asincrónicas con `IAsyncEnumerable<T>`

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:

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    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)
    {
```

```
        foreach (string word in line.Split(' ',  
StringSplitOptions.RemoveEmptyEntries))  
        {  
            yield return word;  
        }  
  
        line = await readStream.ReadLineAsync();  
    }  
}
```

En el ejemplo anterior, las líneas de una cadena se leen de forma asíncrona. 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 asíncrona la línea siguiente de la cadena de origen.

Vea también

- [FromResult](#)
- [Procesamiento de tareas asíncronas a medida que se completan](#)
- [Programación asíncrona con `async` y `await` \(C#\)](#)
- [`async`](#)
- [`await`](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Iniciar varias tareas asincrónicas y procesarlas a medida que se completan (C#)

Artículo • 23/05/2023

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.

Prerrequisitos

Puede seguir este tutorial mediante una de las opciones siguientes:

- [Visual Studio 2022](#) con la carga de trabajo de **desarrollo de escritorio de .NET** instalada. El SDK de .NET se instala automáticamente al seleccionar esta carga de trabajo.
- [SDK de .NET](#) con un editor de código de su elección, como [Visual Studio Code](#).

Creación de una aplicación de ejemplo

Cree una nueva aplicación de consola de .NET Core. Puede crear una mediante el comando [dotnet new console](#) o desde Visual Studio.

Abra el archivo *Program.cs* en el editor de código y reemplace el código existente por este:

```
C#  
  
using System.Diagnostics;  
  
namespace ProcessTasksAsTheyFinish;  
  
class Program  
{  
    static void Main(string[] args)
```

```
{  
    Console.WriteLine("Hello World!");  
}  
}
```

Adición de campos

Dentro de la definición de la clase `Program`, agregue los dos campos siguientes:

C#

```
static readonly HttpClient s_client = new HttpClient  
{  
    MaxResponseContentBufferSize = 1_000_000  
};  
  
static readonly IEnumerable<string> s_urlList = new string[]  
{  
    "https://learn.microsoft.com",  
    "https://learn.microsoft.com/aspnet/core",  
    "https://learn.microsoft.com/azure",  
    "https://learn.microsoft.com/azure/devops",  
    "https://learn.microsoft.com/dotnet",  
    "https://learn.microsoft.com/dynamics365",  
    "https://learn.microsoft.com/education",  
    "https://learn.microsoft.com/enterprise-mobility-security",  
    "https://learn.microsoft.com/gaming",  
    "https://learn.microsoft.com/graph",  
    "https://learn.microsoft.com/microsoft-365",  
    "https://learn.microsoft.com/office",  
    "https://learn.microsoft.com/powershell",  
    "https://learn.microsoft.com/sql",  
    "https://learn.microsoft.com/surface",  
    "https://learn.microsoft.com/system-center",  
    "https://learn.microsoft.com/visualstudio",  
    "https://learn.microsoft.com/windows",  
    "https://learn.microsoft.com/xamarin"  
};
```

`HttpClient` expone la capacidad de enviar solicitudes HTTP y de recibir respuestas HTTP. `s_urlList` contiene todas las direcciones URL que planea procesar la aplicación.

Actualización del punto de entrada de la aplicación

El punto de entrada principal de la aplicación de consola es el método `Main`. Reemplace el método existente por lo siguiente:

C#

```
static Task Main() => SumPageSizesAsync();
```

El método `Main` actualizado ahora se considera un método `Async main`, el cual permite un punto de entrada asincrónico en el archivo ejecutable. Se expresa como una llamada a `SumPageSizesAsync`.

Creación de un método `SumPageSizes` asincrónico

Agregue el método `SumPageSizesAsync` después del método `Main`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}
```

El bucle `while` quita una de las tareas de cada iteración. Una vez completadas todas las tareas, el bucle finaliza. El método comienza creando una instancia e iniciando una clase `Stopwatch`. Después, 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:

```
C#  
  
IEnumerable<Task<int>> downloadTasksQuery =  
    from url in s_urlList  
    select ProcessUrlAsync(url, s_client);
```

Debido a la ejecución diferida con LINQ, se llama a `Enumerable.ToList` para iniciar cada tarea.

```
C#  
  
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

El bucle `while` realiza los pasos siguientes para cada tarea de la colección:

1. Espera una llamada a `WhenAny` para identificar la primera tarea de la colección que ha finalizado su descarga.

```
C#  
  
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. Quita la tarea de la colección.

```
C#  
  
downloadTasks.Remove(finishedTask);
```

3. Espera `finishedTask`, que se devuelve mediante una llamada a `ProcessUrlAsync`.

La variable `finishedTask` es un `Task<TResult>` donde `TResult` 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`, en lugar de leer la propiedad `Task<TResult>.Result` que iniciaría la excepción `AggregateException`.

```
C#  
  
total += await finishedTask;
```

Adición de un método de proceso

Agregue el siguiente método `ProcessUrlAsync` después del método `SumPageSizesAsync`:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Para cualquier dirección URL, el método usará la instancia de `client` proporcionada para obtener la respuesta como `byte[]`. La longitud se devuelve después de que la dirección URL y la longitud se escriban en la consola.

Ejecute el programa varias veces para comprobar que las longitudes que se han descargado no aparecen siempre en el mismo orden.

⊗ Precaución

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 *Program.cs* para el ejemplo.

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
```

```

        "https://learn.microsoft.com/aspnet/core",
        "https://learn.microsoft.com/azure",
        "https://learn.microsoft.com/azure/devops",
        "https://learn.microsoft.com/dotnet",
        "https://learn.microsoft.com/dynamics365",
        "https://learn.microsoft.com/education",
        "https://learn.microsoft.com/enterprise-mobility-security",
        "https://learn.microsoft.com/gaming",
        "https://learn.microsoft.com/graph",
        "https://learn.microsoft.com/microsoft-365",
        "https://learn.microsoft.com/office",
        "https://learn.microsoft.com/powershell",
        "https://learn.microsoft.com/sql",
        "https://learn.microsoft.com/surface",
        "https://learn.microsoft.com/system-center",
        "https://learn.microsoft.com/visualstudio",
        "https://learn.microsoft.com/windows",
        "https://learn.microsoft.com/xamarin"
    };

    await SumPageSizesAsync();

    async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        IEnumerable<Task<int>> downloadTasksQuery =
            from url in s_urlList
            select ProcessUrlAsync(url, s_client);

        List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

        int total = 0;
        while (downloadTasks.Any())
        {
            Task<int> finishedTask = await Task.WhenAny(downloadTasks);
            downloadTasks.Remove(finishedTask);
            total += await finishedTask;
        }

        stopwatch.Stop();

        Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
        Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client)
    {
        byte[] content = await client.GetByteArrayAsync(url);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }

    // Example output:

```

```
// https://learn.microsoft.com 132,517
// https://learn.microsoft.com/powershell 57,375
// https://learn.microsoft.com/gaming 33,549
// https://learn.microsoft.com/aspnet/core 88,714
// https://learn.microsoft.com/surface 39,840
// https://learn.microsoft.com/enterprise-mobility-security 30,903
// https://learn.microsoft.com/microsoft-365 67,867
// https://learn.microsoft.com/windows 26,816
// https://learn.microsoft.com/xamarin 57,958
// https://learn.microsoft.com/dotnet 78,706
// https://learn.microsoft.com/graph 48,277
// https://learn.microsoft.com/dynamics365 49,042
// https://learn.microsoft.com/office 67,867
// https://learn.microsoft.com/system-center 42,887
// https://learn.microsoft.com/education 38,636
// https://learn.microsoft.com/azure 421,663
// https://learn.microsoft.com/visualstudio 30,925
// https://learn.microsoft.com/sql 54,608
// https://learn.microsoft.com/azure/devops 86,034

// Total bytes returned: 1,454,184
// Elapsed time: 00:00:01.1290403
```

Vea también

- [WhenAny](#)
- [Programación asíncrona con async y await \(C#\)](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Acceso asincrónico a archivos (C#)

Artículo • 10/05/2023

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.

Uso de las clases adecuadas

Los ejemplos sencillos de este tema muestran cómo usar los métodos `File.WriteAllTextAsync` y `File.ReadAllTextAsync`. Para tener control finito sobre las operaciones de E/S de archivos, use 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 del grupo de subprocesos 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](#). Las llamadas asincrónicas son más rápidas en aplicaciones de interfaz de usuario aunque un subproceso del grupo de subprocesos se bloquee, porque el subproceso de interfaz de usuario no se bloquea durante la espera.

Escritura de texto

En los siguientes ejemplos 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`. El modificador `async` se encuentra en la definición de métodos que usan la instrucción `await`.

Ejemplo sencillo

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Ejemplo de control finito

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
```

```
 FileMode.Create, 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:

C#

```
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`.

Lectura de texto

En los ejemplos siguientes se lee texto de un archivo.

Ejemplo sencillo

C#

```
public async Task SimpleReadAsync()  
{  
    string filePath = "simple.txt";  
    string text = await File.ReadAllTextAsync(filePath);  
  
    Console.WriteLine(text);  
}
```

Ejemplo de control finito

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 [Task<Int32>](#), por lo que la evaluación de `await` genera un valor `Int32 numRead` una vez completada la operación. Para obtener más información, consulte [Tipos de valor devueltos asincrónicos \(C#\)](#).

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var 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 los siguientes ejemplos se muestra el procesamiento paralelo escribiendo 10 archivos de texto.

Ejemplo sencillo

```
C#  
  
public async Task SimpleParallelWriteAsync()  
{  
    string folder = Directory.CreateDirectory("tempfolder").Name;  
    IList<Task> writeTaskList = new List<Task>();  
  
    for (int index = 11; index <= 20; ++ index)  
    {  
        string fileName = $"file-{index:00}.txt";  
        string filePath = $"{folder}/{fileName}";  
        string text = $"In file {index}{Environment.NewLine}";  
  
        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));  
    }  
  
    await Task.WhenAll(writeTaskList);  
}
```

Ejemplo de control finito

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.

Cualquier aumento del rendimiento se debe casi por completo al procesamiento en paralelo y no al procesamiento asincrónico. Las ventajas de la asincronía radican en que no inmoviliza varios subprocesos ni el subproceso de interfaz de usuario.

```
C#  
  
public async Task ProcessMultipleWritesAsync()  
{  
    IList<FileStream> sourceStreams = new List<FileStream>();  
  
    try  
    {  
        string folder = Directory.CreateDirectory("tempfolder").Name;  
        IList<Task> writeTaskList = new List<Task>();  
  
        for (int index = 1; index <= 10; ++ index)
```

```

    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";

        string text = $"In file {index}{Environment.NewLine}";
        byte[] encodedText = Encoding.Unicode.GetBytes(text);

        var sourceStream =
            new FileStream(
                filePath,
                FileMode.Create, FileAccess.Write, FileShare.None,
                bufferSize: 4096, useAsync: true);

        Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
        sourceStreams.Add(sourceStream);

        writeTaskList.Add(writeTask);
    }

    await Task.WhenAll(writeTaskList);
}
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 más información, consulte [Cancelación de subprocessos administrados](#).

Vea también

- [Programación asíncrona con async y await \(C#\)](#)
- [Tipos de valor devueltos asíncronicos \(C#\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Cancelar una lista de tareas

Artículo • 28/03/2023

Puede cancelar una aplicación de consola asíncrona si no quiere esperar a que termine. Mediante el ejemplo de este tema, puede agregar una cancelación a una aplicación que descargue el contenido de una lista de sitios web. Puede cancelar muchas tareas asociando la instancia de [CancellationTokenSource](#) a cada tarea. Si se presiona la tecla [Entrar](#), se cancelan todas las tareas que aún no se han completado.

Este tutorial abarca lo siguiente:

- ✓ Creación de una aplicación de consola de .NET
- ✓ Escritura de una aplicación asíncrona que admite la cancelación
- ✓ Demostración de la señalización de una cancelación

Requisitos previos

Este tutorial requiere lo siguiente:

- [.NET 5 o un SDK posterior](#)
- Entorno de desarrollo integrado (IDE)
 - [Se recomienda Visual Studio o Visual Studio Code](#)

Creación de una aplicación de ejemplo

Cree una nueva aplicación de consola de .NET Core. Puede crear una mediante el comando [dotnet new console](#) o desde [Visual Studio](#). Abra el archivo *Program.cs* en su editor de código favorito.

Reemplazo de instrucciones using

Reemplace las instrucciones using existentes por estas declaraciones:

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Adición de campos

Dentro de la definición de la clase `Program`, agregue estos tres campos:

C#

```
static readonly CancellationTokenSource s_cts = new  
CancellationTokenSource();  
  
static readonly HttpClient s_client = new HttpClient  
{  
    MaxResponseContentBufferSize = 1_000_000  
};  
  
static readonly IEnumerable<string> s_urlList = new string[]  
{  
    "https://learn.microsoft.com",  
    "https://learn.microsoft.com/aspnet/core",  
    "https://learn.microsoft.com/azure",  
    "https://learn.microsoft.com/azure/devops",  
    "https://learn.microsoft.com/dotnet",  
    "https://learn.microsoft.com/dynamics365",  
    "https://learn.microsoft.com/education",  
    "https://learn.microsoft.com/enterprise-mobility-security",  
    "https://learn.microsoft.com/gaming",  
    "https://learn.microsoft.com/graph",  
    "https://learn.microsoft.com/microsoft-365",  
    "https://learn.microsoft.com/office",  
    "https://learn.microsoft.com/powershell",  
    "https://learn.microsoft.com/sql",  
    "https://learn.microsoft.com/surface",  
    "https://learn.microsoft.com/system-center",  
    "https://learn.microsoft.com/visualstudio",  
    "https://learn.microsoft.com/windows",  
    "https://learn.microsoft.com/xamarin"  
};
```

`CancellationTokenSource` se usa para indicar una cancelación solicitada a un token de cancelación (`CancellationToken`). `HttpClient` expone la capacidad de enviar solicitudes HTTP y de recibir respuestas HTTP. `s_urlList` contiene todas las direcciones URL que planea procesar la aplicación.

Actualización del punto de entrada de la aplicación

El punto de entrada principal de la aplicación de consola es el método `Main`. Reemplace el método existente por lo siguiente:

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}
```

El método `Main` actualizado ahora se considera un método `Async main`, el cual permite un punto de entrada asincrónico en el archivo ejecutable. Escribe algunos mensajes informativos en la consola y, luego, declara una instancia de `Task` denominada `cancelTask`, la cual leerá las pulsaciones de teclas de la consola. Si se presiona la tecla `Entrar`, se realiza una llamada a `CancellationTokenSource.Cancel()`. Esto indicará la cancelación. Después, se asigna la variable `sumPageSizesTask` desde el método `SumPageSizesAsync` y ambas tareas se pasan a `Task.WhenAny(Task[])`, que continuará cuando se complete cualquiera de las dos tareas.

El siguiente bloque de código garantiza que la aplicación no salga hasta que se haya procesado la cancelación. Si la primera tarea que se va a completar es `cancelTask`, se suspende `sumPageSizeTask` con `await`. Si se ha cancelado, cuando se esperaba que se hubiera suspendido con `await`, produce una excepción `System.Threading.Tasks.TaskCanceledException`. El bloque detecta esa excepción e imprime un mensaje.

Creación de un método SumPageSizes asincrónico

Agregue el método `SumPageSizesAsync` después del método `Main`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}
```

El método comienza creando una instancia e iniciando una clase `Stopwatch`. Luego, recorre en bucle cada dirección URL en `s_urlList` y llama a `ProcessUrlAsync`. Con cada iteración, se pasa el token `s_cts.Token` al método `ProcessUrlAsync` y el código devuelve una clase `Task<TResult>`, donde `TResult` es un entero:

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Adición de un método de proceso

Agregue el siguiente método `ProcessUrlAsync` después del método `SumPageSizesAsync`:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Para cualquier dirección URL, el método usará la instancia de `client` proporcionada para obtener la respuesta como `byte[]`. La instancia de `CancellationToken` se pasa a los métodos `HttpClient.GetAsync(String, CancellationToken)` y `HttpContent.ReadAsByteArrayAsync()`. El token (`token`) se usa para registrar la cancelación solicitada. La longitud se devuelve después de que la dirección URL y la longitud se escriban en la consola.

Ejemplo de resultado de la aplicación

Consola

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com                                37,357
https://learn.microsoft.com/aspnet/core                      85,589
https://learn.microsoft.com/azure                           398,939
https://learn.microsoft.com/azure/devops                   73,663
https://learn.microsoft.com/dotnet                         67,452
https://learn.microsoft.com/dynamics365                  48,582
https://learn.microsoft.com/education                     22,924
```

ENTER key pressed: cancelling downloads.

Application ending.

Ejemplo completo

El código siguiente es el texto completo del archivo `Program.cs` para el ejemplo.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
    CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://learn.microsoft.com",
        "https://learn.microsoft.com/aspnet/core",
        "https://learn.microsoft.com/azure",
        "https://learn.microsoft.com/azure/devops",
        "https://learn.microsoft.com/dotnet",
        "https://learn.microsoft.com/dynamics365",
        "https://learn.microsoft.com/education",
        "https://learn.microsoft.com/enterprise-mobility-security",
        "https://learn.microsoft.com/gaming",
        "https://learn.microsoft.com/graph",
        "https://learn.microsoft.com/microsoft-365",
        "https://learn.microsoft.com/office",
        "https://learn.microsoft.com/powershell",
        "https://learn.microsoft.com/sql",
        "https://learn.microsoft.com/surface",
        "https://learn.microsoft.com/system-center",
        "https://learn.microsoft.com/visualstudio",
        "https://learn.microsoft.com/windows",
        "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }
        });

        Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
        s_cts.Cancel();
    });
}
```

```

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel
request was processed.");
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
}

```

Vea también

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Programación asincrónica con async y await \(C#\)](#)

Pasos siguientes

Cancelar tareas asincrónicas tras un período de tiempo (C#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cancelación de tareas asincrónicas tras un período de tiempo

Artículo • 12/09/2023

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 el tutorial [Cancelación de una lista de tareas \(C#\)](#) para descargar una lista de sitios web y mostrar la longitud del contenido de cada uno de ellos.

Este tutorial abarca lo siguiente:

- ✓ Actualización de una aplicación de consola .NET existente
- ✓ Programación de una cancelación

Requisitos previos

Este tutorial requiere lo siguiente:

- Que haya creado una aplicación en el tutorial [Cancelación de una lista de tareas \(C#\)](#)
- [.NET 5 o un SDK posterior](#)
- Entorno de desarrollo integrado (IDE)
 - [Se recomienda Visual Studio o Visual Studio Code](#)

Actualización del punto de entrada de la aplicación

Reemplace el método `Main` existente por lo siguiente:

```
C#  
  
static async Task Main()  
{  
    Console.WriteLine("Application started.");  
  
    try  
    {
```

```
s_cts.CancelAfter(3500);

    await SumPageSizesAsync();
}
catch (OperationCanceledException)
{
    Console.WriteLine("\nTasks cancelled: timed out.\n");
}
finally
{
    s_cts.Dispose();
}

Console.WriteLine("Application ending.");
}
```

El método actualizado `Main` escribe algunos mensajes informativos en la consola. En la instrucción `try-catch`, una llamada a `CancellationTokenSource.CancelAfter(Int32)` programa una cancelación. Esto indicará la cancelación pasado un período de tiempo.

Después, se espera al método `SumPageSizesAsync`. Si el procesamiento de todas las direcciones URL se produce más rápido que la cancelación programada, la aplicación finaliza. Pero si la cancelación programada se desencadena antes de que se procesen todas las direcciones URL, se produce una excepción `OperationCanceledException`.

Ejemplo de resultado de la aplicación



```
Consola

Application started.

https://learn.microsoft.com 37,357
https://learn.microsoft.com/aspnet/core 85,589
https://learn.microsoft.com/azure 398,939
https://learn.microsoft.com/devops 73,663

Tasks cancelled: timed out.

Application ending.
```

Ejemplo completo

El código siguiente es el texto completo del archivo `Program.cs` para el ejemplo.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
    CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://learn.microsoft.com",
        "https://learn.microsoft.com/aspnet/core",
        "https://learn.microsoft.com/azure",
        "https://learn.microsoft.com/azure/devops",
        "https://learn.microsoft.com/dotnet",
        "https://learn.microsoft.com/dynamics365",
        "https://learn.microsoft.com/education",
        "https://learn.microsoft.com/enterprise-mobility-security",
        "https://learn.microsoft.com/gaming",
        "https://learn.microsoft.com/graph",
        "https://learn.microsoft.com/microsoft-365",
        "https://learn.microsoft.com/office",
        "https://learn.microsoft.com/powershell",
        "https://learn.microsoft.com/sql",
        "https://learn.microsoft.com/surface",
        "https://learn.microsoft.com/system-center",
        "https://learn.microsoft.com/visualstudio",
        "https://learn.microsoft.com/windows",
        "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }
    }

    static async Task SumPageSizesAsync()
    {
        var totalSize = 0L;
        var cancellation = s_cts.Token;
        var client = s_client;
        var urls = s_urlList;
        var tasks = new List<Task>();

        foreach (var url in urls)
        {
            tasks.Add(client.GetFromJsonAsync<PageSize>(url));
        }

        await Task.WhenAll(tasks);
    }
}
```

```

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
        Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
}

```

Vea también

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Programación asincrónica con async y await \(C#\)](#)
- [Cancelación de una lista de tareas \(C#\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

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

Artículo • 28/03/2023

Las **secuencias asincrónicas** modelan un origen de datos de streaming. Las secuencias de datos suelen recuperar o generar elementos de forma asincrónica. 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 el equipo para que ejecute .NET, incluido el compilador de C#. El compilador de C# está disponible con [Visual Studio 2022](#) o el [SDK de .NET](#).

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.

Advertencia

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 conoce bien C# y .NET, incluidos Visual Studio o la CLI de .NET.

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 [asynchronous-programming/snippets](#).

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:

C#

```
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 `GetEnvVariable` 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`:

C#

```
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 asíncrona 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. Sigue la solicitud de 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 firma 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:

C#

```
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:

C#

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

Reemplace esas tres líneas por el código siguiente:

C#

```
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:

C#

```
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:

C#

```

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`:

C#

```

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:

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
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:

C#

```
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");

            J0bject results =
J0bject.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 (J0bject issue in issues(results)[ "nodes" ]!)
                yield return issue;
        }

        J0bject issues(J0bject result) => (J0bject)result[ "data" ]!
[ "repository" ]![ "issues" ]!;
        J0bject pageInfo(J0bject result) => (J0bject)issues(result)
[ "pageInfo" ]!;
    }
}

```

El atributo [System.Runtime.CompilerServices.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:

C#

```

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 [asynchronous-programming/snippets](#).

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 asíncrona 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.

En este tutorial, ha usado flujos asíncronos para leer elementos individuales de una API de red que devuelve páginas de datos. Los flujos asíncronos también pueden leer "flujos que nunca terminan", como un teletipo de bolsa o un sensor. La llamada a `MoveNextAsync` devuelve el siguiente elemento en cuanto está disponible.

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



.NET feedback

The .NET documentation is open
source. Provide feedback here.

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Tipos de referencia que aceptan valores NULL

Artículo • 04/02/2024

En un contexto que admite un valor NULL y es "oblivious", todos los tipos de referencia aceptaban valores NULL. Los *tipos de referencia que aceptan valores NULL* hacen referencia a un grupo de características habilitadas en un contexto que admite un valor NULL que minimizan las probabilidades de que el código haga que el tiempo de ejecución genere [System.NullReferenceException](#). Los *tipos de referencia que aceptan valores NULL* incluyen tres características que ayudan a evitar estas excepciones, incluida la capacidad de marcar explícitamente un tipo de referencia como que *acepta valores NULL*:

- Análisis de flujo estático mejorado que determina si una variable puede ser `null` antes de desreferenciarlo.
- Los atributos que anotan las API para que el análisis de flujo determine el *null-state*.
- Las anotaciones de variables que los desarrolladores usan para declarar explícitamente el *null-state* previsto para una variable.

El compilador realiza un seguimiento del *estado NULL* de cada expresión del código en tiempo de compilación. El *estado NULL* tiene uno de estos tres valores:

- *not-null*: se sabe que la expresión no es `null`.
- *maybe-null*: la expresión podría ser `null`.
- *oblivious*: el compilador no puede determinar si la expresión es NULL.

Las anotaciones de variables determinan la *nulabilidad* de una variable de tipo de referencia:

- *non-nullable*: si asigna un valor `null` o una expresión *maybe-null* a la variable, el compilador emite una advertencia. Las variables *non-nullable* (que no admiten un valor NULL) tienen un estado predeterminado NULL de *not-null*.
- *Nullable*: puede asignar un valor `null` o una expresión *maybe-null* a la variable. Cuando el estado NULL de la variable sea *maybe-null*, el compilador emitirá una advertencia si desreferencia la variable. El estado NULL predeterminado de la variable es *maybe-null*.
- *oblivious*: puede asignar un valor `null` o una expresión *maybe-null* a la variable. El compilador no emite advertencias al desreferenciar la variable o al asignar una expresión *maybe-null* a la variable.

El estado `NULL` *oblivious* y la nulabilidad *oblivious* coinciden con el comportamiento antes de que se introdujeran tipos de referencia que admiten valores `NULL`. Esos valores son útiles durante la migración o cuando la aplicación usa una biblioteca que no ha habilitado tipos de referencia que aceptan valores `NULL`.

El análisis de `null-state` y las anotaciones de variables están deshabilitados de forma predeterminada en los proyectos existentes, lo que significa que todos los tipos de referencia siguen aceptando valores `NULL`. A partir de .NET 6, están habilitados de forma predeterminada en los proyectos *nuevos*. Para obtener información sobre cómo habilitar estas características mediante la declaración de un *contexto de anotación que acepta valores NULL*, vea [Contextos que aceptan valores NULL](#).

En el resto de este artículo se describe cómo funcionan esas tres áreas de características para generar advertencias cuando el código pueda estar **desreferenciando** un valor `null`. Desreferenciar una variable significa acceder a uno de sus miembros mediante el operador `.` (punto), como se muestra en el ejemplo siguiente:

```
C#
```

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

Al desreferenciar una variable cuyo valor es `null`, el entorno de ejecución produce una excepción `System.NullReferenceException`.

Obtendrá información sobre:

- [Análisis de estado `NULL`](#) del compilador: cómo determina el compilador si una expresión es `not-null` o `maybe-null`.
- [Atributos](#) que se aplican a las API que proporcionan más contexto para el análisis de estado `NULL` del compilador.
- [Anotaciones de variables que admiten valores `NULL`](#) que proporcionan información sobre la intención de las variables. Las anotaciones son útiles para que los campos establezcan el estado `NULL` predeterminado al principio de los métodos miembro.
- Las reglas que rigen [los argumentos de tipo genérico](#). Se agregaron nuevas restricciones porque los parámetros de tipo pueden ser tipos de referencia o tipos de valor. El sufijo `?` se implementa de forma diferente para los tipos de valor que admiten un valor `NULL` y los tipos de referencia que admiten un valor `NULL`.
- [Los contextos que admiten un valor `NULL`](#) le ayudan a migrar proyectos grandes. Puede habilitar contextos o advertencias que admiten valores `NULL` en partes de la aplicación a medida que migre. Después de solucionar más advertencias, puede habilitar los tipos de referencia que admiten valores `NULL` para todo el proyecto.

Por último, descubrirá los problemas conocidos para el análisis de estado NULL en matrices y tipos `struct`.

También puede explorar estos conceptos en nuestro módulo de aprendizaje sobre [Seguridad sobre la aceptación de valores NULL en C#](#).

Análisis del estado NULL

Cuando se habilitan los tipos de referencia que aceptan valores NULL, *el análisis de estado NULL* realiza un seguimiento del *estado NULL* de las referencias. Una expresión puede ser *not-null* o *maybe-null*. El compilador determina que una variable es *not-null* de dos maneras:

1. Se ha asignado un valor a la variable que se sabe que *not-null*.
2. La variable se ha comprobado con `null` y no se ha modificado desde esa comprobación.

Cuando los tipos de referencia que admiten valores NULL no están habilitados, todas las expresiones tienen el estado NULL de *oblivious*. En el resto de la sección se describe el comportamiento cuando se habilitan los tipos de referencia que admiten valores NULL.

Cualquier variable que el compilador no haya determinado como *not-null* se considera *maybe-null*. El análisis proporciona advertencias en situaciones en las que puede desreferenciar accidentalmente un valor `null`. El compilador genera advertencias basadas en el estado *null-state*.

- Cuando una variable es *not-null*, esa variable se puede desreferenciar de forma segura.
- Cuando una variable es *maybe-null*, se debe comprobar esa variable para asegurarse de que no sea `null` antes de desreferenciarla.

Considere el ejemplo siguiente:

C#

```
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not-null.
Console.WriteLine($"The length of the message is {message.Length}");
```

```
// warning!
Console.WriteLine(originalMessage.Length);
```

En el ejemplo anterior, el compilador determina que `message` es *maybe-null* cuando se imprime el primer mensaje. No hay ninguna advertencia para el segundo mensaje. La línea de código final genera una advertencia porque `originalMessage` podría ser `NULL`. En el ejemplo siguiente se muestra un uso más práctico para recorrer un árbol de nodos hasta la raíz y procesar cada nodo durante el recorrido:

C#

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

El código anterior no genera advertencias para desreferenciar la variable `current`. El análisis estático determina que `current` nunca se desreferencie cuando es *maybe-null*. La variable `current` se comprueba con `null` antes de acceder a `current.Parent` y antes de pasar `current` a la acción `ProcessNode`. En los ejemplos anteriores se muestra cómo el compilador determina el estado *null-state* de las variables locales cuando se inicializan, se asignan o se comparan con `null`.

El análisis del estado `NULL` no realiza un seguimiento de los métodos llamados. Como resultado, los campos inicializados en un método auxiliar común llamado por todos los constructores generarán una advertencia con la plantilla siguiente:

La propiedad "name" que no acepta valores `NULL` debe contener un valor distinto de `NULL` al salir del constructor.

Puede solucionar estas advertencias de una de estas dos maneras: *encadenamiento de constructores* o *atributos que aceptan valores `NULL`* en el método auxiliar. En el código siguiente se muestra un ejemplo de cada caso. La clase `Person` usa un constructor común al que llaman todos los demás constructores. La clase `Student` tiene un método auxiliar anotado con el atributo

[System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#):

C#

```

using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}

```

① Nota

Se agregaron varias mejoras para la asignación definitiva y el análisis de estado NULL en C# 10. Al actualizar a C# 10, es posible que encuentre menos advertencias que admiten un valor NULL que son falsos positivos. Puede obtener más

información sobre las mejoras en la especificación de características para las mejoras de la asignación definitiva.

El análisis de estado que acepta valores NULL y las advertencias que genera el compilador ayudan a evitar errores de programa mediante la anulación de la referencia a `null`. En el artículo sobre [cómo resolver advertencias que admiten valores NULL](#) se proporcionan técnicas para corregir las advertencias que probablemente verá en el código.

Atributos en firmas de API

El análisis del estado NULL necesita sugerencias de los desarrolladores para comprender la semántica de las API. Algunas API proporcionan comprobaciones NULL y deben cambiar el estado *null-state* de una variable de *maybe-null* a *not-null*. Otras API devuelven expresiones que son *not-null* o *maybe-null* en función del estado *null-state* de los argumentos de entrada. Por ejemplo, considere el siguiente código que muestra un mensaje en mayúsculas:

C#

```
void PrintMessageUpper(string? message)
{
    if (!IsNull(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message.ToUpper()}");
    }
}

bool IsNull(string? s) => s == null;
```

En función de la inspección, cualquier desarrollador consideraría que este código es seguro y no debería generar advertencias. Sin embargo, el compilador no sabe que `IsNull` proporciona una comprobación de NULL y emitirá una advertencia para la instrucción `message.ToUpper()`, considerando `message` como una variable *maybe-null*. Use el atributo [NotNullWhen](#) para corregir esta advertencia:

C#

```
bool IsNull([NotNullWhen(false)] string? s) => s == null;
```

Este atributo informa al compilador de que, si `IsNull` devuelve `false`, el parámetro `s` no es NULL. Esto permite al compilador cambiar el *estado NULL* de `message` a *not-null*.

dentro del bloque `if (!IsNull(message)) {...}`. No se emiten advertencias.

Los atributos proporcionan información detallada sobre el estado NULL de los argumentos, los valores devueltos y los miembros de la instancia de objeto utilizada para invocar a un miembro. Los detalles de cada atributo se pueden encontrar en el artículo de referencia del lenguaje sobre los [atributos de referencia que aceptan valores NULL](#). A partir de .NET 5, se anotan todas las API en tiempo de ejecución de .NET. Para mejorar el análisis estático, se anotan las API para proporcionar información semántica sobre el estado *null-state* de los argumentos y los valores devueltos.

Anotaciones de variables que aceptan valores NULL

El análisis del *estado NULL* proporciona un análisis sólido para las variables locales. El compilador necesita más información de usted para las variables de miembro. El compilador necesita más información para establecer el *estado NULL* de todos los campos en el corchete de apertura de un miembro. Cualquiera de los constructores accesibles se podría usar para inicializar el objeto. Si un campo de miembro se puede establecer alguna vez en `null`, el compilador debe suponer que su *null-state* es *maybe-null* al principio de cada método.

Se usan anotaciones que pueden declarar si una variable es un **tipo de referencia que acepta valores NULL** o un **tipo de referencia que no acepta valores NULL**. Estas anotaciones hacen instrucciones importantes sobre el estado *null-state* de las variables:

- **Se supone que una referencia no debe ser NULL.** El estado predeterminado de una variable de referencia que no admite valores NULL es *not-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. El compilador emite una advertencia cuando el código asigna una expresión *maybe-null* a una variable que no debería ser NULL.
- **Una referencia puede ser nula.** El estado predeterminado de una variable de referencia que acepta valores NULL es *maybe-null*. El compilador aplica reglas para garantizar que haya comprobado correctamente una referencia `null`:
 - Solo se puede desreferenciar la variable si el compilador puede 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.

- El compilador no emite advertencias cuando el código asigna una expresión *maybe-null* a una variable que podría ser NULL.

Cualquier variable de referencia que no acepta valores NULL tiene un *estado NULL* predeterminado de *not-null*. Cualquier variable de referencia nula tiene el *estado NULL* inicial o *maybe-null*.

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`:

C#

```
string? name;
```

Cuando se habilitan los tipos de referencia que aceptan valores NULL, 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 cuando se habilita esta característica. Sin embargo, cualquier variable local con tipo implícito (declarada mediante `var`) es un **tipo de referencia que acepta valores NULL**. Como se ha mostrado en las secciones anteriores, el análisis estático determina el *estado NULL* de las variables locales para determinar si son *maybe-null* antes de desreferenciarlo.

En algunas ocasiones, debe invalidar una advertencia si sabe que una variable no es NULL pero el compilador determina que su *null-state* es *maybe-null*. Use el **operador null-forgiving** `!` después de un nombre de variable para forzar que *null-state* sea *not-null*. 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:

C#

```
name!.Length;
```

Los tipos de referencia que aceptan valores NULL y los tipos de valor que aceptan valores NULL proporcionan un concepto semántico similar: una variable puede representar un valor u objeto, o esa variable puede ser `null`. Sin embargo, los tipos de referencia que aceptan valores NULL y los tipos de valor que aceptan valores NULL se implementan de forma diferente: los tipos de valor que aceptan valores NULL se implementan mediante [System.Nullable<T>](#) y los tipos de referencia que aceptan

valores NULL se implementan mediante atributos leídos por el compilador. Por ejemplo, `string?` y `string` se representan mediante el mismo tipo: `System.String`. Sin embargo, `int?` y `int` se representan mediante `System.Nullable<System.Int32>` y `System.Int32`, respectivamente.

Los tipos de referencia no nulas son una característica de tiempo de compilación. Esto significa que es posible que los autores de llamadas ignoren las advertencias, utilizando intencionadamente `null` como argumento de un método que espera una referencia no nula. Los autores de bibliotecas deben incluir comprobaciones en tiempo de ejecución con valores de argumento NULL. `ArgumentNullException.ThrowIfNull` es la opción preferida para comparar un parámetro con NULL en tiempo de ejecución.

ⓘ Importante

La habilitación de anotaciones nulas puede cambiar la forma en que Entity Framework Core determina si se requiere un miembro de datos. Puede obtener más información en el artículo sobre [aspectos básicos de Entity Framework: Trabajar con tipos de referencia nula](#).

Genéricos

Los genéricos requieren reglas detalladas para controlar `T?` para cualquier parámetro de tipo `T`. Las reglas se detallan necesariamente debido al historial y a la implementación diferente para un tipo de valor que acepta valores NULL y un tipo de referencia que acepta valores NULL. [Los tipos de valor que aceptan valores NULL](#) se implementan mediante la estructura `System.Nullable<T>`. [Los tipos de referencia que aceptan valores NULL](#) se implementan como anotaciones de tipo que proporcionan reglas semánticas al compilador.

- Si el argumento de tipo de `T` es un tipo de referencia, `T?` hace referencia al tipo de referencia que acepta valores NULL correspondiente. Por ejemplo, si `T` es un elemento `string`, entonces `T?` es un elemento `string?`.
- Si el argumento de tipo de `T` es un tipo de valor, `T?` hace referencia al mismo tipo de valor, `T`. Por ejemplo, si `T` es un elemento `int`, `T?` también es un elemento `int`.
- Si el argumento de tipo de `T` es un tipo de referencia que acepta valores NULL, `T?` hace referencia a ese mismo tipo de referencia que acepta valores NULL. Por ejemplo, si `T` es un elemento `string?`, entonces `T?` también es un elemento `string?`.

- Si el argumento de tipo de `T` es un tipo de valor que acepta valores NULL, `T?` hace referencia a ese mismo tipo de valor que acepta valores NULL. Por ejemplo, si `T` es un elemento `int?`, entonces `T?` también es un elemento `int?`.

Para los valores devueltos, `T?` es equivalente a `[MaybeNull]T`; para los valores de argumento, `T?` es equivalente a `[AllowNull]T`. Para obtener más información, consulte el artículo sobre [Atributos para el análisis de estado NULL](#) en la referencia del lenguaje.

Puede especificar un comportamiento diferente mediante [restricciones](#):

- La restricción `class` significa que `T` debe ser un tipo de referencia que no acepta valores NULL (por ejemplo, `string`). El compilador genera una advertencia si se usa un tipo de referencia que acepta valores NULL, como `string?` para `T`.
- La restricción `class?` significa que `T` debe ser un tipo de referencia, ya sea un tipo de referencia que no acepta valores NULL (`string`) o un tipo de referencia que acepta valores NULL (por ejemplo, `string?`). Cuando el parámetro de tipo es un tipo de referencia que acepta valores NULL, como `string?`, una expresión de `T?` hace referencia a ese mismo tipo de referencia que acepta valores NULL, como `string?`.
- La restricción `notnull` significa que `T` debe ser un tipo de referencia que no acepta valores NULL o un tipo de valor que no acepta valores NULL. Si usa un tipo de referencia que acepta valores NULL o un tipo de valor que acepta valores NULL para el parámetro de tipo, el compilador genera una advertencia. Además, cuando `T` es un tipo de valor, el valor devuelto es ese tipo de valor, no el tipo de valor que acepta valores NULL correspondiente.

Estas restricciones ayudan a proporcionar más información al compilador sobre cómo se usará `T`. Esto es útil cuando los desarrolladores eligen el tipo para `T` y proporciona un mejor análisis de *estado NULL* cuando se usa una instancia del tipo genérico.

Contextos que aceptan valores NULL

Para proyectos pequeños, puede habilitar tipos de referencia que aceptan valores NULL, corregir advertencias y continuar. Sin embargo, para proyectos más grandes y soluciones de varios proyectos, esto podría generar un gran número de advertencias. Puede usar pragmas para habilitar los tipos de referencia que aceptan valores NULL archivo a archivo a medida que comienza a usar tipos de referencia que aceptan valores NULL. Las nuevas características que protegen contra la generación de un elemento `System.NullReferenceException` pueden ser perjudiciales cuando están activadas en un código base existente:

- Todas las variables de referencia con tipo explícito se interpretan como tipos de referencia que no aceptan valores NULL.
- El significado de la restricción `class` en genéricos cambió para significar un tipo de referencia que no acepta valores NULL.
- Se generan nuevas advertencias debido a estas nuevas reglas.

El **contexto de anotación que acepta valores NULL** determina el comportamiento del compilador. Hay cuatro valores para el **contexto de anotación que acepta valores NULL**:

- *disable*: el código es "*oblivious*" que admite un valor NULL. Deshabilitar coincide con el comportamiento antes de que se habilitaran los tipos de referencia que aceptan valores NULL, excepto que la nueva sintaxis genera advertencias en lugar de errores.
 - Las advertencias que aceptan valores NULL están deshabilitadas.
 - Todas las variables de tipo de referencia son tipos de referencia que aceptan valores NULL.
 - El uso del sufijo `?` para declarar un tipo de referencia que acepta valores NULL genera una advertencia.
 - Puede usar el operador que permite un valor NULL, `!`, pero no tiene ningún efecto.
- *enable*: el compilador permite todo el análisis de referencias nulas y todas las características del lenguaje.
 - Todas las nuevas advertencias que aceptan valores NULL están habilitadas.
 - Puede usar el sufijo `?` para declarar un tipo de referencia que acepta valores NULL.
 - Las variables de tipo de referencia sin el sufijo `?` son tipos de referencia que no aceptan valores NULL.
 - El operador que permite valores NULL elimina las advertencias de una posible asignación a `null`.
- *warnings*: el compilador realiza todos los análisis de valores NULL y emite advertencias cuando el código pueda desreferenciar `null`.
 - Todas las nuevas advertencias que aceptan valores NULL están habilitadas.
 - El uso del sufijo `?` para declarar un tipo de referencia que acepta valores NULL genera una advertencia.
 - Todas las variables de tipo de referencia pueden ser NULL. Sin embargo, los miembros tienen el estado *null-state* de *not-null* en la llave de apertura de todos los métodos, a menos que se declaren con el sufijo `?`.
 - Puede usar el operador que permite valores NULL, `!`.

- *anotaciones*: el compilador no emite advertencias cuando el código puede desreferenciar `null` o cuando se asigna una expresión maybe-null a una variable que no acepta valores NULL.
 - Todas las nuevas advertencias que aceptan valores NULL están deshabilitadas.
 - Puede usar el sufijo `?` para declarar un tipo de referencia que acepta valores NULL.
 - Las variables de tipo de referencia sin el sufijo `?` son tipos de referencia que no aceptan valores NULL.
 - Puede usar el operador que permite un valor NULL, `!`, pero no tiene ningún efecto.

Tanto el contexto de anotación que acepta valores NULL como el contexto de advertencia que acepta valores NULL pueden establecerse en 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. En la tabla siguiente se muestran los valores permitidos y se resumen los contextos que dichos valores especifican.

[+] Expandir tabla

Context	Advertencias de desreferenciación	Advertencias de asignación	Tipos de referencia	Sufijo <code>?</code>	Operador <code>!</code>
<code>disable</code>	Disabled	Disabled	Todos aceptan valores NULL.	Genera una advertencia.	No tiene ningún efecto.
<code>enable</code>	habilitado	habilitado	No acepta valores NULL a menos que se declare con <code>?</code> .	Declara un tipo que acepta valores NULL.	Suprime las advertencias relativas a una posible asignación <code>null</code> .
<code>warnings</code>	habilitado	No aplicable	Todos aceptan valores NULL, pero los miembros se consideran <i>not-null</i> en la llave de	Genera una advertencia.	Suprime las advertencias relativas a una posible asignación <code>null</code> .

Context	Advertencias de desreferenciación	Advertencias de asignación	Tipos de referencia	Sufijo ?	Operador !
			apertura de los métodos		
annotations	Disabled	Disabled	No acepta valores NULL a menos que se declare con ?.	Declara un tipo que acepta valores NULL.	No tiene ningún efecto.

Las variables de tipo de referencia en el código compilado en un contexto *deshabilitado* son *nullable-oblivious*. Puede asignar un valor literal `null` o una variable *maybe-null* a una variable que *admita un valor NULL "oblivious"*. Sin embargo, el estado predeterminado de una variable *nullable-oblivious* es *not-null*.

Puede elegir qué configuración es la mejor para el proyecto:

- Elija *disable* para los proyectos heredados que no quiere actualizar en función de diagnósticos o nuevas características.
- Elija *advertencias* para determinar dónde el código puede producir [System.NullReferenceException](#). Puede solucionar esas advertencias antes de modificar el código para habilitar tipos de referencia que no aceptan valores NULL.
- Elija *annotations* para expresar la intención de diseño antes de habilitar las advertencias.
- Elija *enable* para nuevos proyectos y proyectos activos en los que quiera protegerse de excepciones de referencia nula.

Ejemplo:

XML

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

También puede usar directivas para establecer los mismos contextos en cualquier lugar del código fuente: Estas directivas son más útiles cuando se va a migrar un código base grande.

- `#nullable enable`: establece el contexto de anotación nula y el contexto de advertencia nula en **enable**.
- `#nullable disable`: establece el contexto de anotación nula y el contexto de advertencia nula en **disable**.

- `#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`: establezca el contexto de advertencia nula en **disable**.
- `#nullable enable warnings`: establece el contexto de advertencia nula en **enable**.
- `#nullable restore warnings`: restaura el contexto de advertencia que acepta valores NULL según la configuración del proyecto.
- `#nullable disable annotations`: establece el contexto de anotación nula en **disable**.
- `#nullable enable annotations`: establece el contexto de anotación nula en **enable**.
- `#nullable restore annotations`: restaura el contexto de advertencia de anotación según la configuración del proyecto.

Para cualquier línea de código, puede establecer cualquiera de las siguientes combinaciones:

 Expandir tabla

Contexto de advertencia	Contexto de anotación	Uso
proyecto predeterminado	proyecto predeterminado	Valor predeterminado
enable	disable	Corrección de advertencias de análisis
enable	proyecto predeterminado	Corrección de advertencias de análisis
proyecto predeterminado	enable	Adición de anotaciones de tipo
enable	enable	Código ya migrado
disable	enable	Anotación de código antes de corregir advertencias
disable	disable	Adición de código heredado al proyecto migrado
proyecto predeterminado	disable	Raramente
disable	proyecto predeterminado	Raramente

Esas nueve combinaciones proporcionan un control preciso sobre los diagnósticos que el compilador emite para el código. Puede habilitar más características en cualquier área que esté actualizando sin ver advertencias adicionales que aún no está listo para abordar.

ⓘ Importante

El contexto global que admite un valor NULL no se aplica a los archivos de código generado. En cualquier estrategia, el contexto que admite un valor NULL está **deshabilitado** para cualquier archivo de código fuente marcado como generado. Esto significa que las API de los archivos generados no se anotan. Hay cuatro maneras de marcar un archivo como generado:

1. En el archivo `.editorconfig`, especifique `generated_code = true` en una sección que se aplique a ese archivo.
2. Coloque `<auto-generated>` o `<auto-generated/>` en un comentario en la parte superior del archivo. Puede estar en cualquier línea de ese comentario, pero el bloque de comentario debe ser el primer elemento del archivo.
3. Inicie el nombre de archivo con `TemporaryGeneratedFile_`
4. Finalice el nombre de archivo con `.designer.cs`, `.generated.cs`, `.g.cs` o `.g.i.cs`.

Los generadores pueden optar por usar la directiva de preprocesador `#nullable`.

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. A partir de .NET 6, los proyectos nuevos incluyen el elemento `<Nullable>enable</Nullable>` en todas las plantillas de proyecto.

Estas opciones proporcionan dos estrategias distintas para [actualizar un código base existente](#) para usar tipos de referencia que aceptan valores NULL.

Problemas conocidos

Las matrices y estructuras que contienen tipos de referencia son dificultades conocidas en las referencias que aceptan valores NULL y el análisis estático que determina la seguridad de los valores NULL. En ambas situaciones, se puede inicializar una referencia que no acepta valores NULL en `null` sin generar advertencias.

Estructuras

Una estructura que contiene tipos de referencia que no aceptan valores NULL permite asignarle `default` sin ninguna advertencia. Considere el ejemplo siguiente:

```
C#  
  
using System;  
  
#nullable enable  
  
public struct Student  
{  
    public string FirstName;  
    public string? MiddleName;  
    public string LastName;  
}  
  
public static class Program  
{  
    public static void PrintStudent(Student student)  
    {  
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");  
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");  
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");  
    }  
  
    public static void Main() => PrintStudent(default);  
}
```

En el ejemplo anterior, no hay ninguna advertencia en `PrintStudent(default)` mientras que los tipos de referencia que no aceptan valores NULL `FirstName` y `LastName` son NULL.

Otro caso más común es cuando se trata de estructuras genéricas. Considere el ejemplo siguiente:

```
C#  
  
#nullable enable  
  
public struct S<T>  
{  
    public T Prop { get; set; }  
}  
  
public static class Program  
{  
    public static void Main()  
    {  
        string s = default(S<string>).Prop;
```

```
    }  
}
```

En el ejemplo anterior, la propiedad `Prop` es `null` en tiempo de ejecución. Se asigna a una cadena que no acepta valores NULL sin ninguna advertencia.

Matrices

Las matrices también son un problema conocido en los tipos de referencia que aceptan valores NULL. Consideré el ejemplo siguiente, que no genera ninguna advertencia:

```
C#  
  
using System;  
  
#nullable enable  
  
public static class Program  
{  
    public static void Main()  
    {  
        string[] values = new string[10];  
        string s = values[0];  
        Console.WriteLine(s.ToUpper());  
    }  
}
```

En el ejemplo anterior, la declaración de la matriz muestra que contiene cadenas que no aceptan valores NULL, mientras que todos sus elementos se inicializan en `null`.

Después, a la variable `s` se le asigna un valor `null` (el primer elemento de la matriz). Por último, se desreferencia la variable `s`, lo que genera una excepción en tiempo de ejecución.

Vea también

- [Tipos de referencia que aceptan valores NULL: propuesta](#)
- [Borrador de especificación de tipos de referencia que aceptan valores NULL](#)
- [Anotaciones de parámetros de tipo sin restricciones](#)
- [Tutorial de introducción a las referencias que no aceptan valores NULL](#)
- [Nullable \(opción del compilador de C#\)](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Actualización de un código base con tipos de referencia que admiten un valor NULL para mejorar las advertencias sobre diagnósticos nulos

Artículo • 07/04/2023

Los [tipos de referencia que admiten un valor NULL](#) permiten declarar si a las variables de un tipo de referencia se les debe asignar o no un valor `null`. El análisis estático y las advertencias del compilador cuando el código podría desreferenciar `null` son la ventaja más importante de esta característica. Una vez habilitado, el compilador genera advertencias que ayudan a evitar que se genere una excepción `System.NullReferenceException` cuando se ejecuta el código.

Si el código base es relativamente pequeño, puede activar la [característica en el proyecto](#), resolver advertencias y disfrutar de las ventajas que ofrecen los diagnósticos mejorados. Los códigos base más grandes pueden requerir un enfoque más estructurado para resolver advertencias a lo largo del tiempo, habilitar la característica para algunos a medida que usted resuelve advertencias en distintos tipos o archivos. En este artículo se describen distintas estrategias para actualizar un código base y las contrapartidas asociadas a estas estrategias. Antes de iniciar la migración, lea la introducción conceptual de los [tipos de referencia que admiten un valor NULL](#). En ella se trata el análisis estático del compilador, los valores *null-state* de *maybe-null* y *not-null* y las anotaciones que admiten un valor NULL. Una vez que se haya familiarizado con esos conceptos y términos, podrá migrar su código.

Planeación de la migración

Independientemente de cómo actualice su código base, el objetivo es que las advertencias y anotaciones que admiten un valor NULL estén habilitadas en su proyecto. Una vez que alcance ese objetivo, tendrá el valor `<nullable>Enable</nullable>` en su proyecto. No necesitará ninguna de las directivas del preprocesador para ajustar la configuración en otro lugar.

La primera opción es establecer el valor predeterminado para el proyecto. Las opciones son:

1. **'Disable' que admite valores NULL como valor predeterminado:** `disable` es el predeterminado si no se agrega ningún elemento `Nullable` a su archivo de

proyecto. Use este valor predeterminado cuando no esté agregando activamente nuevos archivos al código base. La actividad principal es actualizar la biblioteca para que utilice tipos de referencia que admiten un valor NULL. Si se usa este valor predeterminado, se agrega una directiva de preprocesador que admite un valor NULL a cada archivo a medida que actualice su código.

2. **'Enable' que admite valores NULL como valor predeterminado:** Establezca este valor predeterminado al desarrollar activamente características nuevas. Quiere que todo el código nuevo se beneficie de los tipos de referencia y análisis estáticos que admiten un valor NULL. Si usa este valor predeterminado, deberá agregar un valor `#nullable disable` en la parte superior de cada archivo. Quitará estas directivas de preprocesador a medida que aborde las advertencias de cada archivo.
3. **Advertencias que admiten valores NULL como valor predeterminado:** Elija este valor predeterminado para una migración de dos fases. En la primera fase, resuelva las advertencias. En la segunda fase, active las anotaciones para declarar el valor *null-state* que se espera de una variable. Si usa este valor predeterminado, deberá agregar un valor `#nullable disable` en la parte superior de cada archivo.
4. **Anotaciones que admiten valores NULL** como valor predeterminado. Anote el código antes de resolver las advertencias.

Al habilitar una opción que admite un valor NULL como predeterminada, se genera más trabajo previo para agregar las directivas del preprocesador a cada archivo. La ventaja es que todos los archivos de código nuevos que se agreguen al proyecto estarán habilitados para aceptar valores NULL. Cualquier trabajo nuevo admitirá valores NULL; solo se debe actualizar el código existente. Si se deshabilita la opción que admite un valor NULL, este proceso funcionará mejor si la biblioteca es estable y el objetivo principal del desarrollo es adoptar tipos de referencia que admiten un valor NULL. Los tipos de referencia que aceptan valores NULL se habilitan al anotar las API. Cuando haya terminado, habilite los tipos de referencia que aceptan valores NULL para todo el proyecto. Al crear un archivo nuevo, debe agregar las directivas del preprocesador y hacer que sea compatible con los valores NULL. Si alguno de los desarrolladores de su equipo se olvida de hacerlo, ese nuevo código se sumará al trabajo pendiente para hacer que todo el código admita valores NULL.

La elección de una estrategia u otra dependerá de la cantidad de desarrollo activo que haya en el proyecto. Cuanto más desarrollado esté y más estable sea su proyecto, más adecuada será la segunda estrategia. Cuantas más características se estén desarrollando, más apropiada será la primera estrategia.

 **Importante**

El contexto global que admite un valor NULL no se aplica a los archivos de código generado. En cualquier estrategia, el contexto que admite un valor NULL está *deshabilitado* para cualquier archivo de código fuente marcado como generado. Esto significa que las API de los archivos generados no se anotan. Hay cuatro maneras de marcar un archivo como generado:

1. En el archivo `.editorconfig`, especifique `generated_code = true` en una sección que se aplique a ese archivo.
2. Coloque `<auto-generated>` o `<auto-generated/>` en un comentario en la parte superior del archivo. Puede estar en cualquier línea de ese comentario, pero el bloque de comentario debe ser el primer elemento del archivo.
3. Inicie el nombre de archivo con `TemporaryGeneratedFile_`
4. Finalice el nombre de archivo con `.designer.cs`, `.generated.cs`, `.g.cs` o `.g.i.cs`.

Los generadores pueden optar por usar la directiva de preprocesador `#nullable`.

Información sobre contextos y advertencias

Si se habilitan las advertencias y las anotaciones, se controlará cómo el compilador visualiza los tipos de referencia y la nulabilidad. Cada tipo tiene una de las tres nulabilidades:

- *oblivious*: todos los tipos de referencia son de tipo *oblivious* que admiten un valor NULL cuando el contexto de anotación de deshabilita.
- *nonnullable*: tipo de referencia no anotado; `c` es *nonnullable* cuando el contexto de anotación se habilite.
- *nullable*: tipo de referencia anotado; `c?` es de tipo *nullable*, pero puede que se genere una advertencia cuando el contexto de anotación se deshabilite. Las variables declaradas con `var` son de tipo *nullable* cuando el contexto de anotación se habilita.

El compilador genera advertencias en función de esa nulabilidad:

- Los tipos *nonnullable* generan advertencias si se les asigna un valor `null` potencial.
- Los tipos *nullable* generan advertencias si se desreferencian en el caso *maybe-null*.
- Los tipos *oblivious* generan advertencias si se desreferencian en el caso *maybe-null* y el contexto de advertencia está habilitado.

Cada variable tiene un estado predeterminado que admite un valor NULL que depende de su nulabilidad:

- Las variables que admiten un valor NULL tienen un estado predeterminado *null-state* de *maybe-null*.
- Las variables que no admiten un valor NULL tienen un estado predeterminado *null-state* de *not-null*.
- Las variables de tipo "oblivious" que admiten un valor NULL tienen un estado predeterminado *null-state* de *not-null*.

Antes de habilitar los tipos de referencia que aceptan valores NULL, todas las declaraciones del código base son de tipo *nullable oblivious*. Esto es importante porque significa que todos los tipos de referencia tienen en estado predeterminado *null-state* de *not-null*.

Resolución de advertencias

Si su proyecto usa Entity Framework Core, debe leer sus guías sobre cómo [trabajar con tipos de referencia que admiten un valor NULL](#).

Al iniciar la migración, debe empezar habilitando solo las advertencias. Todas las declaraciones siguen siendo de tipo *nullable oblivious*, pero se le mostrarán advertencias cuando desreferencie un valor después de que su estado *null-state* cambie a *maybe-null*. A medida que se resuelvan estas advertencias, realizará comprobaciones de NULL en más ubicaciones, y su código base será más resistente. Para obtener información sobre técnicas específicas para diferentes situaciones, vea el artículo sobre [Técnicas para resolver advertencias que admiten un valor NULL](#).

Puede resolver advertencias y habilitar anotaciones en cada archivo o clase antes de continuar con otro código. Sin embargo, a menudo resulta más eficaz resolver las advertencias generadas mientras el contexto es *warnings* para poder habilitar las anotaciones de tipo. De esta forma, todos los tipos serán *oblivious* hasta que haya resuelto el primer conjunto de advertencias.

Habilitación de anotaciones de tipo

Después de resolver el primer conjunto de advertencias, podrá habilitar el *contexto de anotación*. Así, se cambiarán los tipos de referencia de *oblivious* a *nonnullable*. Todas las variables declaradas con `var` son de tipo *nullable*. Con este cambio se suelen generar nuevas advertencias. El primer paso para resolver las advertencias del compilador es usar anotaciones `?` en los tipos de parámetros y de valores devueltos para indicar si los argumentos o los valores devueltos pueden ser `null`. A medida que realiza esta tarea, su objetivo no es solo resolver las advertencias; el objetivo más importante es hacer que el compilador entienda su intención de admitir posibles valores NULL.

Atributos para complementar a las anotaciones de tipo

Se han agregado varios atributos para expresar información adicional sobre el estado `NULL` de las variables. Es probable que las reglas de sus API sean más complicadas que *not-null* o *maybe-null* para todos los parámetros y valores devueltos. Muchas de las API tienen reglas más complejas para cuando las variables pueden ser `null` o no. En estos casos, usará atributos para expresar dichas reglas. Los atributos que describen la semántica de su API se encuentran en el artículo sobre los [Atributos que afectan a los análisis que admiten un valor NULL](#).

Pasos siguientes

Una vez que haya resuelto todas las advertencias después de habilitar las anotaciones, podrá establecer el contexto predeterminado para su proyecto en `enabled`. Si ha agregado alguna pragma a su código para la anotación que admite un valor `NULL` o el contexto de advertencia, la podrá eliminar. Con el tiempo, es posible que se le muestren nuevas advertencias. Puede escribir código que introduzca advertencias. Una dependencia de biblioteca se puede actualizar para los tipos de referencia que aceptan valores `NULL`. Esas actualizaciones cambiarán los tipos de esa biblioteca de *nullable oblivious* a *nonnullable* o *nullable*.

También puede explorar estos conceptos en nuestro módulo de aprendizaje sobre [Seguridad sobre la aceptación de valores NULL en C#](#).

Métodos de C#

Artículo • 24/05/2024

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.

ⓘ Nota

En este tema se analizan los métodos denominados. Para obtener información sobre las funciones anónimas, consulte [Expresiones lambda](#).

Firmas de método

Los métodos se declaran en un elemento `class`, `record` o `struct` al especificar lo siguiente:

- Un nivel de acceso opcional, como, por ejemplo, `public` o `private`. De manera predeterminada, 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.

ⓘ Importante

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:

C#

```

namespace MotorCycleExample
{
    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 diferencian en sus tipos de parámetros.

Invocación de método

Los métodos pueden ser de *instancia* o *estáticos*. Debe crear una instancia de un objeto para invocar un método de instancia en esa instancia; un método de instancia funciona en esa 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 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 se separan mediante 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.

C#

```
public static class SquareExample
{
    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`. El segundo se convierte en el valor del parámetro `speed`.

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var 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.

C#

```
namespace NamedMotorCycle;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed) =>
        (int)Math.Round((double)miles / speed, 0);

    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        int travelTime = moto.Drive(miles: 170, speed: 60);
        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 los argumentos con nombre solo pueden ir detrás de argumentos posicionales si están en la posición correcta. En el ejemplo siguiente se invoca el método `TestMotorcycle.Drive` del ejemplo anterior con un argumento posicional y un argumento con nombre.

C#

```
int 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. El método `Equals`, sin embargo, no se define en la clase `Person`; se hereda de `Object`.

C#

```
public class Person
{
    public string FirstName = default!;
}

public static class ClassTypeExample
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { 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 al 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).

C#

```
namespace methods;

public class Person
{
    public string FirstName = default!;

    public override bool Equals(object? obj) =>
        obj is Person p2 &&
        FirstName.Equals(p2.FirstName);

    public override int GetHashCode() => FirstName.GetHashCode();
```

```
}

public static class Example
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { 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](#). De manera predeterminada, los tipos de valor y los tipos de referencia se pasan por valor a un método.

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.

C#

```
public static class ByValueExample
{
    public static void Main()
    {
        var 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.

C#

```

public class SampleRefType
{
    public int value;
}

public static class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType { 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 vuelve al método de llamada. Para pasar un parámetro por referencia, use las palabras 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.

C#

```
public static class ByRefExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    private 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.

C#

```
public static class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        Console.WriteLine("i = {0}  j = {1}", i, j);
```

```

        Swap(ref i, ref j);

        Console.WriteLine("i = {0}  j = {1}", i, j);
    }

    static void Swap(ref int x, ref int y) =>
        (y, x) = (x, y);
}
// 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.

Colecciones 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 colección 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 colección 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 cuatro maneras siguientes para el parámetro `params`:

- Si se pasa una colección del tipo adecuado que contenga el número de elementos que se quiera. En el ejemplo se usa una [expresión de colección](#) para que el compilador cree un tipo de colección adecuado.
- Si se pasa una lista separada por comas de los argumentos individuales del tipo adecuado para el método. El compilador crea el tipo de colección adecuado.
- Pasando `null`.
- Si no se proporciona un argumento a la colección de parámetros.

En el ejemplo siguiente se define un método denominado `GetVowels` que devuelve todas las vocales de una colección de parámetros. El método `Main` muestra las cuatro 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 una colección vacía.

C#

```

static class ParamsExample
{
    static void Main()

```

```

{
    string fromArray = GetVowels(["apple", "banana", "pear"]);
    Console.WriteLine($"Vowels from collection expression:
'{fromArray}'");

    string fromMultipleArguments = GetVowels("apple", "banana", "pear");
    Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

    string fromNull = GetVowels(null);
    Console.WriteLine($"Vowels from null: '{fromNull}'");

    string fromNoValue = GetVowels();
    Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
}

static string GetVowels(params IEnumerable<string>? input)
{
    if (input == null || !input.Any())
    {
        return string.Empty;
    }

    char[] vowels = ['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: 'aeaaaaea'
//      Vowels from multiple arguments: 'aeaaaaea'
//      Vowels from null: ''
//      Vowels from no value: ''

```

Antes de C# 13, el modificador `params` solo se puede usar con una matriz unidimensional.

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 parámetrosopcionales 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 `default(SomeType)`, donde `SomeType` puede ser un tipo de valor o un tipo de referencia. Si es un tipo de referencia, es efectivamente lo mismo que especificar `null`. Puede usar el literal `default`, ya que el compilador puede inferir el tipo de la declaración del parámetro.
- Una expresión con el formato `new ValType()`, donde `ValType` es un tipo de valor. Esta expresión invoca el constructor sin parámetros implícito del tipo de valor, que no es un miembro real del tipo.

(!) Nota

En C# 10 y versiones posteriores, cuando una expresión con el formato `new ValType()` invoca el constructor sin parámetros definido explícitamente de un tipo de valor, el compilador genera un error, ya que el valor del parámetro predeterminado debe ser una constante en tiempo de compilación. Use la expresión `default(ValType)` o el literal `default` para proporcionar el valor de parámetro predeterminado. Para más información sobre los constructores sin parámetros, consulte la sección [Inicialización de estructuras y valores predeterminados](#) del artículo [Tipos de estructuras](#).

Si un método incluye parámetros necesarios y opcionales, los parámetrosopcionales 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.

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                             string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} =
{required + optionalInt}";
        Console.WriteLine(msg);
    }
}
```

El autor de la llamada debe proporcionar un argumento para todos los parámetros opcionales hasta el último parámetro opcional para el que se proporciona un argumento. Por ejemplo, en el 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 argumentosopcionales, mientras que la segunda omite el último argumento. La tercera llamada de 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`.

C#

```
public static class OptionsExample
{
    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:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

El uso de parámetros opcionales afecta a la *resolución de sobrecarga* o la forma en que el compilador de C# determina qué sobrecarga invocar para una llamada de método, como se indica a continuación:

- Un método, indizador o constructor es un candidato para la ejecución si cada uno de sus parámetros corresponde, por nombre o por posición, a un solo argumento y ese 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.

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 devuelve 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 deja de ejecutarse cuando alcanza el final del bloque de código.

Por ejemplo, estos dos métodos utilizan la palabra clave `return` para devolver enteros:

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2) =>
        number1 + number2;

    public int SquareANumber(int number) =>
        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:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
```

```
// The result is 9.  
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));  
// The result is 9.  
Console.WriteLine(result);
```

A veces, quiere que el método devuelva más que un solo valor. Los *tipos de tupla* y los *literales de tupla* se usan para devolver varios valores. 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 primer nombre, segundo nombre y apellido, junto con la edad, de un objeto `PersonInfo`.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)  
{  
    PersonInfo per = PersonInfo.RetrieveInfoById(id);  
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);  
}
```

A continuación, el autor de la llamada puede consumir la tupla devuelta mediante el código siguiente:

C#

```
var person = GetPersonalInfo("1111111111");  
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:

C#

```
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 `GetPersonalInfo` se puede modificar luego de la manera siguiente:

C#

```
var person = GetPersonalInfo("111111111");  
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Si un método toma una matriz como parámetro y modifica el valor de los elementos individuales, no es necesario que el método devuelva la matriz. 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.

C#

```
public static class ArrayValueExample  
{  
    static void Main()  
    {  
        int[] values = [2, 4, 6, 8];  
        DoubleValues(values);  
        foreach (var value in values)  
        {  
            Console.Write("{0} ", value);  
        }  
    }  
  
    public static void DoubleValues(int[] arr)  
    {  
        for (var ctr = 0; ctr <= arr.GetUpperBound(0); 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. Modificar el origen crea un cambio importante 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 usando la 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 obtener 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.

ⓘ Nota

Un método asincrónico vuelve al autor de la 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 normalmente tiene un tipo de valor devuelto de `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` 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. Un método asincrónico puede tener [cualquier tipo de valor devuelto que sea como 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.

```
C#  
  
class Program  
{  
    static Task Main() => DoSomethingAsync();  
  
    static 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();  
  
        Console.WriteLine($"Result: {result}");  
    }  
  
    static async Task<int> DelayAsync()  
    {  
        await Task.Delay(100);  
        return 5;  
    }  
}  
// Example 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, consulte los artículos [Programación asincrónica con async y await](#) y [Tipos de valor devueltos asincrónicos](#).

Miembros con forma de expresión

Es habitual tener definiciones de método que hacen las devoluciones de manera 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 `=>`:

C#

```
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). En el caso de las propiedades y los indexadores, solo deben leerse, y no se debe usar 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 utiliza 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>`, `IAsyncEnumerable<T>`, `IEnumerator` o `IEnumerator<T>`.

Para obtener más información, consulta [Iteradores](#).

Consulte también

- [Modificadores de acceso](#)
- [Clases estáticas y sus miembros](#)
- [Herencia](#)
- [Clases y miembros de clase abstractos y sellados](#)
- [params](#)
- [out](#)
- [ref](#)
- [in](#)
- [Pasar parámetros](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Iterators

Artículo • 10/05/2023

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* para los elementos de esa clase. Un *iterador* es un objeto que atraviesa un contenedor, especialmente las listas. Los iteradores 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 generar y consumir secuencias. Estas secuencias se pueden generar y consumir de forma sincrónica o asíncrona. Este artículo proporciona información general sobre esas características.

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:

```
C#  
  
foreach (var item in collection)  
{  
    Console.WriteLine(item?.ToString());  
}
```

That's all. (Esto es todo) 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](#).

Cuando se genera una secuencia de forma asíncrona, puede usar la instrucción `await foreach` para consumir la secuencia de forma asíncrona:

```
C#  
  
await foreach (var item in asyncSequence)  
{  
    Console.WriteLine(item?.ToString());  
}
```

Cuando una secuencia es `System.Collections.Generic.IEnumerable<T>`, se usa `foreach`.

Cuando una secuencia es `System.Collections.Generic.IAsyncEnumerable<T>`, se usa `await foreach`. En el último caso, la secuencia se genera de forma asíncrona.

Orígenes de enumeración con métodos de iterador

Otra magnífica característica del lenguaje C# permite generar métodos que crean un origen para una enumeración. Estos métodos 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:

```
C#  
  
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:

```
C#  
  
public IEnumerable<int> GetSingleDigitNumbersLoop()  
{  
    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:

```
C#  
  
public IEnumerable<int> GetSetsOfNumbers()  
{  
    int index = 0;  
    while (index < 10)  
        yield return index++;  
  
    yield return 50;  
  
    index = 100;  
    while (index < 110)  
        yield return index++;  
}
```

Todos estos ejemplos anteriores tendrían un homólogo asincrónico. En cada caso, reemplazaría el tipo de valor devuelto de `IEnumerable<T>` por un elemento `IAsyncEnumerable<T>`. Por ejemplo, el ejemplo anterior tendría la siguiente versión asincrónica:

```
C#  
  
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()  
{  
    int index = 0;  
    while (index < 10)  
        yield return index++;  
  
    await Task.Delay(500);  
  
    yield return 50;  
  
    await Task.Delay(500);
```

```
    index = 100;
    while (index < 110)
        yield return index++;
}
```

Esta es la sintaxis de los iteradores sincrónicos y asincrónicos. Veamos un ejemplo del mundo real. 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:

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence,
int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Si la lectura desde el dispositivo IoT genera una secuencia asincrónica, modificaría el método como se muestra en el método siguiente:

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await 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. El código siguiente no se compilará:

C#

```
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:

C#

```

public IEnumerable<int> GetFirstDecile()
{
    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 cinco primeros números impares, basándose en un argumento booleano. Eso se podría escribir como estos dos métodos:

C#

```

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:

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

El código exacto generado por el compilador es 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:

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
```

```
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

El compilador traduce el primer ejemplo asincrónico en algo similar a esta construcción:

C#

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

La manera en que el enumerador se elimina depende de las características del tipo de `enumerator`. En el caso sincrónico general, la cláusula `finally` se expande a:

C#

```
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

El caso asincrónico general se expande a:

C#

```
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

Sin embargo, si el tipo de `enumerator` es un tipo sellado y no hay conversión implícita del tipo de `enumerator` a `IDisposable` o `IAsyncDisposable`, la cláusula `finally` se expande en un bloque vacío:

```
C#
```

```
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:

```
C#
```

```
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 delegados y eventos en C#

Artículo • 10/05/2023

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 tiene seguridad de tipos. El compilador garantiza que los tipos coincidan con los argumentos y los tipos de valor devuelto.

Se han agregado [punteros de función](#) a C# 9 para escenarios similares, donde se necesita más control sobre la convención de llamadas. El código asociado a un delegado se invoca mediante un método virtual agregado a un tipo de delegado. Mediante los punteros de función puede especificar otras convenciones.

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. Los delegados permiten 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 de método. Verá 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 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 serie tratarán sobre las características del lenguaje, la compatibilidad con bibliotecas y las expresiones comunes que se usan al trabajar con los delegados y eventos. Obtendrá información sobre:

- La palabra clave `delegate` y el código que genera.
- Las características de la clase `System.Delegate` y la manera en que se usan estas características.
- Cómo crear delegados con seguridad de tipos.
- Cómo crear métodos que se puedan invocar mediante delegados.
- Cómo trabajar con delegados y eventos mediante expresiones lambda.
- Cómo los delegados se convierten en uno de los bloques de creación para LINQ.
- Cómo los delegados son la base del patrón de eventos de .NET y en qué se diferencian.

Comencemos.

[Siguiente](#)

System.Delegate y la palabra clave delegate

Artículo • 10/05/2023

[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:

```
C#  
  
// 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.

 Nota

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.

C#

```
// 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:

C#

```
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:

C#

```
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()`:

C#

```
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:

C#

```
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:

C#

```
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

Artículo • 10/05/2023

[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 flexible 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:

C#

```
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:

C#

```
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 en un valor único:

C#

```
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:

C#

```
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

Artículo • 07/04/2023

[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:

C#

```
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:

C#

```
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 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.

Creación de 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.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            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:

```
C#
```

```
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:

```
C#
```

```
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 el marco de trabajo principal 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.

Formato de salida

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:

```
C#  
  
public enum Severity  
{  
    Verbose,  
    Trace,  
    Information,  
    Warning,  
    Error,  
    Critical  
}
```

```
C#  
  
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}";  
        if (WriteMessage is not null)  
            WriteMessage(outputMsg);  
    }  
}
```

A continuación, vamos a usar ese argumento `Severity` para filtrar los mensajes que se envían a la salida del registro.

```
C#  
  
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}";  
        if (WriteMessage is not null)  
            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.

Creación de 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:

C#

```
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:

C#

```
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:

C#

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteLine += 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:

C#

```
Logger.WriteLine -= 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.

Ninguna parte del código de la clase `Logger` tendrá que actualizarse para admitir cualquiera de los escenarios.

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()`:

C#

```
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. Al usar delegados en el diseño, estos distintos componentes están acoplados débilmente. Esto ofrece varias ventajas. Es 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 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

Artículo • 10/05/2023

[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:

- Permite un acoplamiento mínimo entre un origen de eventos y un receptor de eventos. Estos dos componentes pueden no ser escritos 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`:

C#

```
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:

C#

```
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 `+=`:

C#

```
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 `-=`:

C#

```
fileLister.Progress -= onProgress;
```

Es importante que declare 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

Artículo • 10/05/2023

[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:

C#

```
void EventRaised(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 [sección siguiente](#) que esta convención ya no se aplica). Si el tipo de evento no necesita ningún argumento adicional, seguirá proporcionando ambos 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:

C#

```
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 llenar la clase `FileSearcher` para buscar archivos que coincidan con un patrón y generar el evento correcto cuando se detecte una coincidencia.

C#

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            RaiseFileFound(file);
        }
    }
}
```

```
    private void RaiseFileFound(string file) =>
        FileFound?.Invoke(this, new FileFoundArgs(file));
}
```

Definición y generación de eventos similares a 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:

C#

```
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:

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

y quitar controlador:

C#

```
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 de 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 `EventArgs` 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`:

C#

```
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 `Boolean`, 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:

C#

```
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

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:

C#

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Adición de 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.

C#

```
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.

C#

```
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:

C#

```
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)  
        {  
            RaiseSearchDirectoryChanged(dir, totalDirs, completedDirs++);  
            // Search 'dir' and its subdirectories for files that match the  
search pattern:  
            SearchDirectory(dir, searchPattern);  
        }  
        // Include the Current Directory:  
        RaiseSearchDirectoryChanged(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))  
    {  
        FileFoundArgs args = RaiseFileFound(file);  
        if (args.CancelRequested)
```

```

        {
            break;
        }
    }

private void RaiseSearchDirectoryChanged(
    string directory, int totalDirs, int completedDirs) =>
    _directoryChanged?.Invoke(
        this,
        new SearchDirectoryArgs(directory, totalDirs, completedDirs));

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}

```

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 `DirectoryChanged`, 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.

```
C#
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.

Vea también

- [Introducción a los eventos](#)
- [Diseño de eventos](#)
- [Control y generación de eventos](#)

Más adelante verá algunos cambios en estos patrones en la versión más reciente de .NET.

Siguiente

Patrón de eventos actualizado de .NET Core

Artículo • 09/05/2023

[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 `FileEventArgs` y `SearchDirectoryEventArgs` para que no deriven de `EventArgs`. El programa funcionará exactamente igual.

También puede cambiar `SearchDirectoryEventArgs` a un struct si realiza un cambio más:

C#

```
internal struct SearchDirectoryEventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryEventArgs(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 `FileEventArgs` 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

Todavía debe aprender un último patrón: 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:

C#

```
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 tiene una ejecución continua porque es `async`). El comportamiento real del entorno de ejecución se definirá de forma diferente para distintos entornos. Se puede terminar el subprocesso o el proceso que posee el subprocesso, 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

Artículo • 09/05/2023

[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 llamando 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 donde necesita implementar la devolución de llamada. 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.

Control de versiones en C#

Artículo • 30/09/2024

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.

Versión de lenguaje

El compilador de C# forma parte del SDK de .NET. De forma predeterminada, el compilador elige la versión del lenguaje C# que coincide con el [TFM](#) elegido para su proyecto. Si la versión del SDK es mayor que el marco elegido, el compilador podría usar una versión de lenguaje mayor. Puede cambiar el valor predeterminado si establece el elemento `LangVersion` en su proyecto. Puede obtener información sobre cómo hacerlo en nuestro artículo sobre [opciones del compilador](#).

Advertencia

No se recomienda establecer el elemento `LangVersion` como `latest`. La configuración `latest` significa que el compilador instalado usa su versión más reciente. Esto puede cambiar de equipo a equipo, lo que hace que las compilaciones no sean confiables. Además, habilita las características de lenguaje que pueden requerir tiempo de ejecución o características de biblioteca que no se incluyen en el SDK actual.

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, por ejemplo, versiones preliminares, 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 métodoopcionales anteriores sean obligatorios o cambien su valor predeterminado, se tendrá que actualizar todo el código que no proporcione esos argumentos.

ⓘ Nota

Hacer que los argumentos obligatorios sean opcionales debe tener un efecto muy pequeño, especialmente si no cambia el comportamiento del método.

Cuanto 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.

XML

```
<dependentAssembly>
  <assemblyIdentity name="ReferencedLibrary"
publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
  <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

ⓘ Nota

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:

C#

```
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

Consola

```
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`.

C#

```
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

Consola

```
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#)

Artículo • 11/04/2024

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.](#)
- [Cree un nuevo método para un tipo enum mediante métodos de extensión.](#)

Miembros de clase, registro y estructura

Para implementar un programa se usan clases, registros y estructuras. Estas técnicas suelen usarse al escribir clases, registros 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 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 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 subprocessos 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 subprocessos.

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`.

Comentarios

¿Le ha resultado útil esta página?



[Proporcionar comentarios sobre el producto ↗](#)

Procedimiento para separar cadenas mediante String.Split en C#

Artículo • 07/11/2024

El método [String.Split](#) crea una matriz de subcadenas mediante la división de la cadena de entrada en función de uno o varios delimitadores. A menudo, este método es la manera más fácil de separar una cadena en límites de palabras. También sirve para dividir las cadenas en otras cadenas o caracteres específicos.

ⓘ Nota

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#.

Ejemplos de String.Split

Este código divide una frase común en una matriz de cadenas para cada palabra.

C#

```
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. Dado que las matrices de C# tienen un índice cero, cada cadena de la matriz se indexa de 0 al valor devuelto por la propiedad [Array.Length](#) menos 1:

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' '');
```

```
for (int i = 0; i < words.Length; i++)
{
    System.Console.WriteLine($"Index {i}: <{words[i]}>");
```

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.

C#

```
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 como caracteres de separación, que se pasan a [Split](#) en una matriz. En el bucle al final del código se muestra cada una de las palabras de la matriz devuelta.

C#

```
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:

```
C#  
  
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).

```
C#  
  
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);  
}
```

Uso de GitHub Copilot para dividir una cadena

Puede usar GitHub Copilot en el IDE para generar código para dividir cadenas mediante `String.Split` en C#.

Si usa la versión 17.8 o posterior de Visual Studio 2022, puede probar el Copilot de GitHub controlado por [IA en Visual Studio](#) para generar código para dividir una cadena de entrada en subcadenas basadas en uno o varios delimitadores. Envíe su pregunta

como solicitud en la ventana de chat de Copilot, como en el ejemplo siguiente. También puede enviar mensajes mediante el [chat en línea](#) en la propia ventana del editor.

ⓘ Nota

GitHub Copilot funciona con IA, por lo que es posible que se produzcan sorpresas y errores. Asegúrese de comprobar cualquier código o sugerencia generado. Para obtener más información sobre el uso general de GitHub Copilot, impacto en el producto, supervisión humana y privacidad, consulte [Preguntas más frecuentes sobre GitHub Copilot](#).

En el texto siguiente se muestra un mensaje de ejemplo para Copilot Chat:

Genere código de C# para usar Split.String para dividir una cadena en subcadenas. La cadena de entrada es "Usted gana algo. Pierdes algo". Los delimitadores son espacio y punto. Proporcione una salida de ejemplo.

Puede personalizar la solicitud para usar cadenas y delimitadores según sus requisitos.

Puede usar características de chat, como [comandos de barra diagonal](#), [referencias a archivos](#), [métodos](#), [clases](#) y [subprocesos](#), para establecer la intención y obtener mejores respuestas con contexto con ámbito. Para un archivo existente que está abierto en el IDE, puede preguntar a GitHub Copilot mediante el chat en línea con `/generate code to split string1 in #Filename by delimiter spaces`.

En la salida siguiente se muestra un ejemplo de respuesta de Copilot Chat:

Output

Here is an example of how you can use the Split method in C# to split the string "You win some. You lose some." using space and period as delimiters:

```
using System;

class Program
{
    static void Main()
    {
        string input = "You win some. You lose some.";
        char[] delimiters = new char[] { ' ', '.' };
        string[] substrings = input.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries);

        foreach (string substring in substrings)
        {
            Console.WriteLine(substring);
```

```
        }
    }
}

**Example Output**
You
win
some
You
lose
some
```

Cuando Copilot devuelve un bloque de código, la respuesta incluye opciones para copiar el código, insertar el código en un nuevo archivo u obtener una vista previa de la salida del código.

 **Nota**

Los resultados pueden ser diferentes de lo que se muestra en las respuestas de ejemplo. Los modelos de IA no son deterministas, lo que significa que pueden devolver respuestas diferentes cuando se le haga la misma pregunta. Esto puede deberse al aprendizaje y la adaptación adicionales a lo largo del tiempo, la variación del lenguaje, los cambios en el contexto, como el historial de chat, etc.

GitHub Copilot Chat

e. You lose some." Delimiters are space and period. Provide example output

Generate C# code to use Split.String to split a string into substrings. Input string is "You win some. You lose some." Delimiters are space and period. Provide example output

GitHub Copilot

Here is an example of how you can use the `Split` method in C# to split the string "You win some. You lose some." using space and period as delimiters:

```
using System;

class Program
{
    static void Main()
    {
        string input = "You win some. You lose some.";
        char[] delimiters = new char[] { ' ', '.' };
        string[] substrings = input.Split(delimiters, StringSplitOptions.RemoveEmptyEntries);

        foreach (string substring in substrings)
        {
            Console.WriteLine(substring);
        }
    }
}
```

Insert in new file Preview

Example Output:

```
You
win
some
You
lose
some
```

Preview

❖ How can I check if a string contains a specific substring in C#?

References Was this helpful?

AI suggestions might be inaccurate.

Para más información, vea:

- [Centro de confianza de GitHub Copilot ↗](#)
- [GitHub Copilot en Visual Studio](#)
- [GitHub Copilot en VS Code ↗](#)

Consulte también

- Extracción de elementos de una cadena
- Cadenas
- Expresiones regulares de .NET

Procedimiento para concatenar varias cadenas (Guía de C#)

Artículo • 15/03/2024

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.

ⓘ Nota

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#.

Literales de cadena

En el ejemplo siguiente se divide un literal de cadena larga en cadenas más pequeñas para mejorar la legibilidad en el código fuente. El código concatena las cadenas más pequeñas para crear el literal de cadena larga. Los elementos 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 implicadas.

C#

```
// 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);
```

Operadores + y +=

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:

C#

```
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);
```

Interpolación de cadenas

En algunas expresiones, es más fácil concatenar cadenas mediante la interpolación de cadena, como se muestra en este código:

C#

```
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);
```

ⓘ Nota

En operaciones de concatenación de cadenas, el compilador de C# trata una cadena NULL igual que una cadena vacía.

A partir de C# 10, se puede utilizar la interpolación de cadenas para inicializar una cadena constante cuando todas las expresiones utilizadas para los marcadores de posición son también cadenas constantes.

String.Format

Otro método para concatenar cadenas es [String.Format](#). Este método funciona bien cuando se crea una cadena a partir de un número reducido de cadenas de componente.

StringBuilder

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.

C#

```
// 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](#).

String.Concat o String.Join

Otra opción para combinar cadenas a partir de una colección consiste en usar el método [String.Concat](#). Use el método [String.Join](#) si un delimitador debe separar cadenas

de origen. El código siguiente combina una matriz de palabras usando ambos métodos:

C#

```
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);
```

LINQ y Enumerable.Aggreagte

Por último, puede usar [LINQ](#) y el método [Enumerable.Aggreagte](#) 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 y se agrega un espacio entre cada palabra de la matriz:

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase}
{word}");
System.Console.WriteLine(phrase);
```

Esta opción puede provocar más asignaciones que otros métodos para concatenar colecciones, ya que crea una cadena intermedia para cada iteración. Si la optimización del rendimiento es fundamental, considere la clase [StringBuilder](#) o los métodos [String.Concat](#) o [String.Join](#) para concatenar una colección, en vez de [Enumerable.Aggreagte](#).

Vea también

- [String](#)
- [StringBuilder](#)
- [Cadenas](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Cómo buscar cadenas

Artículo • 10/05/2023

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.

ⓘ Nota

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:

C#

```
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($"Contains \"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**. El valor de enumeración `StringComparison.CurrentCultureIgnoreCase` se usa 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.

C#

```

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 un único símbolo `\`, que en una cadena de C# debe escribirse como `\\\`).

Modelo	Significado
<code>the</code>	busca coincidencias con el texto "the"
<code>(eir)?</code>	busca coincidencias con 0 o 1 apariciones de "eir"
<code>\s</code>	busca coincidencias con un carácter de espacio en blanco

C#

```
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.WriteLine($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
        System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

💡 Sugerencia

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
<code>^</code>	busca coincidencias con el principio de la cadena
<code>\d{3}</code>	busca coincidencias con exactamente 3 caracteres de dígitos
<code>-</code>	busca coincidencias con el carácter “-”
<code>\d{4}</code>	busca coincidencias con exactamente 4 caracteres de dígitos
<code>\$</code>	busca coincidencias con el final de la cadena

C#

```
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.WriteLine($"{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](#)
- [Lenguaje de expresiones regulares: referencia rápida](#)
- [Procedimientos recomendados para el uso de cadenas en .NET](#)

Procedimiento para modificar el contenido de cadenas en C#

Artículo • 09/05/2023

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 demostrar que las cadenas originales y modificadas son instancias distintas, los ejemplos almacenan el resultado en una variable nueva. Al ejecutar cada ejemplo, se puede examinar tanto el objeto `string` original como el objeto `string` nuevo y modificado.

ⓘ Nota

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.

C#

```
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 *immutable* 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 "_":

C#

```
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.

C#

```
// 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:

```
C#  
  
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 coincide 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](#).

```
C#  
  
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.ToUpper(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.

C#

```

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 llenar 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.

C#

```

// 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 la suya, esa persona obtendrá la copia de usted y no esperará que usted modifique la cadena.

Vea también

- Expresiones regulares de .NET
- Lenguaje de expresiones regulares: referencia rápida

Cómo comparar cadenas en C#

Artículo • 20/03/2024

Las cadenas se comparan para responder a una de estas dos preguntas: "¿Son estas dos cadenas iguales?" 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.

Los campos de enumeración [System.StringComparison](#) representan estas opciones:

- **CurrentCulture**: compara cadenas mediante las reglas de ordenación de la referencia cultural y la referencia cultural actual.
- **CurrentCultureIgnoreCase**: compara cadenas mediante las reglas de ordenación de la referencia cultural y la referencia cultural actual, e ignora el uso de mayúsculas y minúsculas de las cadenas que se comparan.
- **InvariantCulture**: compara cadenas mediante reglas de ordenación que distinguen referencias culturales y la referencia cultural invariable.
- **InvariantCultureIgnoreCase**: compara cadenas mediante reglas de ordenación que distinguen referencias culturales, la referencia cultural invariable e ignorando el caso de las cadenas que se comparan.
- **Ordinal**: compara cadenas mediante las reglas de ordenación ordinal (binaria).
- **OrdinalIgnoreCase**: compara cadenas mediante las reglas de ordenación ordinal (binaria) e ignora el uso de mayúsculas y minúsculas de las cadenas que se comparan.

ⓘ Nota

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.Equals`
- `String.Equality` y `String.Inequality`, es decir, los operadores de igualdad `==` y `!=`, respectivamente, realizan una comparación ordinal que distingue entre mayúsculas y minúsculas. `String.Equals` tiene una sobrecarga en la que se puede proporcionar un argumento `StringComparison` para modificar sus reglas de ordenación. En el siguiente ejemplo se muestra que:

C#

```
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.OrdinalIgnoreCase);
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.

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)`. Todas realizan una comparación que distingue mayúsculas de minúsculas. Sin embargo, aunque las pruebas de igualdad realizan una comparación ordinal, los métodos `CompareTo` y `Compare` realizan una comparación lingüística, que tiene en cuenta

la referencia cultural, mediante la referencia cultural actual. Haga que la intención del código sea clara llamando a una sobrecarga que especifica explícitamente el tipo de comparación que se va a realizar.

Comparaciones de ordinales sin distinción entre mayúsculas y minúsculas

Con el método `String.Equals(String, StringComparison)` puede especificar un valor `StringComparison` de `StringComparison.OrdinalIgnoreCase` para una comparación ordinal que no distingue 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`. Estas comparaciones se muestran en el código siguiente:

C#

```
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");
```

Estos métodos utilizan las convenciones de mayúsculas y minúsculas de la cultura `invariable` cuando realizan una comparación ordinal insensible a mayúsculas y minúsculas.

Comparaciones lingüísticas

Muchos métodos de comparación de cadenas (como `String.StartsWith`) usan reglas lingüísticas para la *referencia cultural actual* de forma predeterminada para ordenar sus entradas. Esta comparación lingüística se conoce a veces como "criterio de ordenación de 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 pequeño, por lo que las cadenas "coop" y "co-op" aparecerían una junto a la otra en una ordenación. Es posible que se omitan algunos caracteres de control que no son de impresión. Además, algunos caracteres Unicode pueden ser equivalentes a una secuencia de instancias de `Char`. En el ejemplo siguiente se usa la frase "Ellos bailan en la calle" en alemán. Usa "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".

C#

```
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");
}

```

En Windows, antes de .NET 5, el criterio de ordenación de "cop", "coop" y "co-op" varía 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. Antes de .NET 5, las API de globalización de .NET usaban bibliotecas de [compatibilidad con idiomas nacionales \(NLS\)](#). En .NET 5 y versiones posteriores, las API de globalización de .NET usan [componentes internacionales para bibliotecas de Unicode \(ICU\)](#) ↗, que unifican el comportamiento de globalización de NET en todos los sistemas operativos compatibles.

Comparaciones con referencias culturales específicas

En el ejemplo siguiente se almacenan objetos [CultureInfo](#) para 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":

C#

```

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.

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 el ejemplo siguiente se muestra cómo ordenar una matriz de cadenas mediante la referencia cultural actual:

```
C#  
  
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\nSorted 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.

```
C#  
  
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 el ejemplo siguiente se muestra cómo buscar en la lista ordenada con la misma función de comparación. La función local `ShowWhere` muestra dónde está o debería estar el texto buscado:

C#

```

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.WriteLine("Not found. Sorts between: ");

    if (index == 0)
        Console.WriteLine("beginning of sequence and ");
    else
        Console.WriteLine($"{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](#).

Vea también

- [System.Globalization.CultureInfo](#)
- [System.StringComparer](#)
- [Cadenas](#)
- [Comparación de cadenas](#)
- [Globalizar y localizar aplicaciones](#)



Colaborar con nosotros en
GitHub

El origen de este contenido se
puede encontrar en GitHub,
donde también puede crear y



Comentarios de .NET

.NET es un proyecto de código
abierto. Seleccione un vínculo para
proporcionar comentarios:

revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

 Abrir incidencia con la documentación

 Proporcionar comentarios sobre el producto

Procedimiento para detectar excepciones no compatibles con CLS

Artículo • 10/05/2023

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.

C#

```
// 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);  
    }  
}
```

Consulte también

- [RuntimeWrappedException](#)
- [Excepciones y control de excepciones](#)

Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

[Reflection](#) provides objects (of type [Type](#)) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the [GetType\(\)](#) method - inherited by all types from the [Object](#) base class - to obtain the type of a variable:

① Note

Make sure you add `using System;` and `using System.Reflection;` at the top of your .cs file.

C#

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

ⓘ Note

The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the [`IsAssembly`](#) property. To identify a `protected internal` method, use the [`IsFamilyOrAssembly`](#).

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the [`SerializableAttribute`](#) attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]  
public class SampleClass  
{  
    // Objects of this type can be serialized.  
}
```

A method with the attribute [`DllImportAttribute`](#) is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

① Note

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the [Attributes](#) section of the [C# language specification](#)

Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

C#

```
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

[] [Expand table](#)

Target value	Applies to
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module
<code>field</code>	Field in a class or a struct
<code>event</code>	Event
<code>method</code>	Method or <code>get</code> and <code>set</code> property accessors
<code>param</code>	Method parameters or <code>set</code> property accessor parameters
<code>property</code>	Property
<code>return</code>	Return value of a method, property indexer, or <code>get</code> property accessor
<code>type</code>	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an [automatically implemented property](#).

The following example shows how to apply attributes to assemblies and modules. For more information, see [Common Attributes \(C#\)](#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

ⓘ Note

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see [AttributeUsage](#).

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the [DllImportAttribute](#) class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the article [Dynamically Loading and Using Types](#).

Related sections

For more information:

- [Common Attributes \(C#\)](#)
- [Caller Information \(C#\)](#)
- [Attributes](#)
- [Reflection](#)
- [View Type Information](#)
- [Reflection and Generic Types](#)
- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

Creación de atributos personalizados

Artículo • 18/03/2023

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:

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct)  
]  
public class AuthorAttribute : System.Attribute  
{  
    private string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
        Version = 1.0;  
    }  
}
```

El nombre de la clase [AuthorAttribute](#) es el nombre del atributo, [Author](#), al que se le agrega el sufijo [Attribute](#). 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:

C#

```
[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.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct,  
                      AllowMultiple = true) // Multiuse attribute.  
]  
public class AuthorAttribute : System.Attribute  
{  
    string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
  
        // Default value.  
        Version = 1.0;  
    }  
  
    public string GetName() => Name;  
}
```

En el ejemplo de código siguiente se aplican varios atributos del mismo tipo a una clase.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]  
public class ThirdClass  
{  
    // ...  
}
```

Vea también

- [System.Reflection](#)
- [Escribir atributos personalizados](#)
- [AttributeUsage \(C#\)](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Acceso a atributos mediante reflexión

Artículo • 18/03/2023

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 muchas versiones sobrecargadas. Para obtener más información, vea [Attribute](#).

Una especificación de atributo como:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

Es conceptualmente equivalente al código siguiente:

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    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`. 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.

Aquí tenemos un ejemplo completo. Se define un atributo personalizado, se aplica a varias entidades y se recupera mediante reflexión.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
    System.AttributeTargets.Struct,
```

```

        AllowMultiple = true) // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => 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
{
    public 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 {t}");

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);
    }
}

```

```
// Displaying output.
foreach (System.Attribute attr in attrs)
{
    if (attr is AuthorAttribute a)
    {
        System.Console.WriteLine($" {a.GetName()}, version
{a.Version:f}");
    }
}
/* 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](#)
- [Recuperar la información almacenada en atributos](#)



Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

[Proporcionar comentarios sobre el producto](#)

Procedimiento para crear una unión de C o C++ mediante atributos en C#

Artículo • 10/04/2023

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`.

En este segmento de código, todos los campos de `TestUnion` empiezan en la misma ubicación en la memoria.

C#

```
[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;
}
```

El código a continuación es otro ejemplo en el que los campos empiezan en ubicaciones diferentes establecidas explícitamente.

C#

```
[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 int i3;
}
```

```
public double d;

[System.Runtime.InteropServices.FieldOffset(12)]
public char c;

[System.Runtime.InteropServices.FieldOffset(14)]
public byte b;
}
```

Los dos campos enteros, `i1` y `i2` combinados, tiene las mismas ubicaciones en la memoria que `lg`. O bien `lg` usa los primeros 8 bytes o `i1` usa los primeros 4 bytes y `i2` usa los 4 bytes siguientes. Este tipo de control sobre el diseño del struct es útil cuando se usa la invocación de plataforma.

Vea también

- [System.Reflection](#)
- [Attribute](#)
- [Atributos](#)



Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

[Proporcionar comentarios sobre el producto](#)

Genéricos y atributos

Artículo • 18/03/2023

Los atributos pueden aplicarse a los tipos genéricos de la misma manera que los tipos no genéricos. Sin embargo, solo puede aplicar atributos en *tipos genéricos abiertos* y *tipos genéricos construidos cerrados*, no en *tipos genéricos construidos parcialmente*. Un *tipo genérico abierto* es uno en el que no se especifica ninguno de los argumentos de tipo, como `Dictionary< TKey, TValue >`. Un *tipo genérico construido cerrado* especifica todos los argumentos de tipo, como `Dictionary<string, object>`. Un *tipo genérico construido parcialmente* especifica algunos argumentos de tipo, pero no todos. Un ejemplo es `Dictionary<string, TValue>`.

En los ejemplos siguientes se usa este atributo personalizado:

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

Un atributo puede hacer referencia a un tipo genérico abierto:

C#

```
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:

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<, >))]
class ClassB { }
```

Un atributo puede hacer referencia a un tipo genérico construido cerrado:

C#

```
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 provoca un error en tiempo de compilación:

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
CS0416
class ClassD<T> { }
```

A partir de C# 11, un tipo genérico puede heredar de [Attribute](#):

C#

```
public class CustomGenericAttribute<T> : Attribute { } //Requires C# 11
```

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

- [Genéricos](#)
- [Atributos](#)



Colaborar con nosotros en
[GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

[Proporcionar comentarios sobre el producto](#)

Procedimiento para consultar los metadatos de un ensamblado con la función de reflexión (LINQ)

Artículo • 18/03/2023

Las API de reflexión de .NET se usan para examinar los metadatos de un ensamblado .NET y para crear colecciones de tipos, miembros de tipo y parámetros 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 de enumeración, como matrices.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
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);
    }
}
```

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.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Genéricos y reflexión

Artículo • 18/03/2023

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 tiempo de ejecución](#).

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](#):

- [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.
- [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](#):

- [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.

Consulte también

- [Genéricos](#)
- [Reflexión y tipos genéricos](#)
- [Genéricos](#)



Colaborar con nosotros en
[GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

nuestra guía para
colaboradores.

☒ Proporcionar comentarios sobre
el producto

Definición y lectura de atributos personalizados

Artículo • 18/03/2023

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. Tenga en cuenta [ObsoleteAttribute](#). 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, obtendrá información sobre cómo agregar atributos al código, cómo crear y usar sus propios atributos y cómo usar algunos atributos que se integran en .NET.

Requisitos previos

Debe configurar la máquina para ejecutar .NET. Puede encontrar las instrucciones de instalación en la página [Descargas de .NET](#). Puede ejecutar esta aplicación en Windows, Ubuntu Linux, macOS o en un contenedor de Docker. Debe instalar el editor de código que prefiera. 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.

Creación de la aplicación

Ahora que ha instalado todas las herramientas, cree una nueva aplicación de consola de .NET. Para usar el generador de línea de comandos, ejecute el siguiente comando en su shell favorito:

CLI de .NET

```
dotnet new console
```

Este comando crea archivos de proyecto de .NET básicos. Ejecute `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-store`.

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.

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`. Este atributo indica que el código está obsoleto y que ya no se debe usar. Puede colocar este atributo en una clase, por ejemplo, mediante corchetes.

```
C#  
  
[Obsolete]  
public class MyClass  
{  
}
```

Mientras que la clase se denomina `ObsoleteAttribute`, solo es necesario usar `[obsolete]` en el código. La mayoría del código de C# sigue esta convención. 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. Se incluye un parámetro de cadena en el atributo Obsoleto para proporcionar esta explicación.

```
C#  
  
[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.

Crear su propio atributo

Para crear un atributo, defina una nueva clase que herede de la clase base `Attribute`.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

Con el código anterior, puede usar `[MySpecial]` (o `[MySpecialAttribute]`) como atributo en otra parte del código base.

C#

```
[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.

Aquí hay un problema que se debe vigilar. Como se mencionó anteriormente, solo se pueden pasar determinados tipos como argumentos al usar atributos. Sin embargo, al crear un tipo de atributo, el compilador de C# no le impide crear esos parámetros. En el ejemplo siguiente, ha creado un atributo con un constructor que se compila correctamente.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
```

```
{  
}  
}
```

Sin embargo, no puede usar este constructor con una sintaxis de atributo.

C#

```
[Gotcha(new Foo(), "test")] // does not compile  
public class AttributeFail  
{  
}
```

El código anterior provoca un error del 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 se pueden usar en los siguientes "destinos". Los ejemplos anteriores los muestran en las clases, pero ahora se pueden usar en:

- Ensamblado
- Clase
- Constructor
- Delegar
- Enumeración
- evento
- Campo
- GenericParameter
- Interfaz
- Método
- Módulo
- Parámetro
- Propiedad.
- ReturnValue
- Estructura

Cuando se crea una clase de atributo, de forma predeterminada, C# permite 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!

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Si intenta colocar el atributo anterior en algo que no sea una clase o una estructura, obtendrá un error del compilador como `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

C#

```
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. La reflexión 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):

C#

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

Se muestra algo parecido a: `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 objeto `MemberInfo`, `FieldInfo` o de otro tipo) puede usar el método `GetCustomAttributes`. Este método devuelve 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 `MethodInfo` para `MyClass` (que anteriormente vimos que contiene un atributo `[Obsolete]`).

C#

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Se muestra lo siguiente 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 crear 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 el entorno de ejecución

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# quita las llamadas a ese método (pero no el método propiamente dicho). Normalmente, se usa esta técnica con fines de depuración (diagnóstico).

- `[CallerMemberName]`. Este atributo se puede usar en parámetros, y reside en el espacio de nombres `System.Runtime.CompilerServices`. `CallerMemberName` es un atributo que se usa para injectar el nombre del método que llama a otro método. Es una manera de eliminar "cadenas mágicas" al implementar `INotifyPropertyChanged` en distintos marcos de interfaz de usuario. Por ejemplo:

C#

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName =
= default!)
    {
        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. El uso de `CallerMemberName` evita los errores relacionados con los tipos y también agiliza los procesos de refactorización o cambio de nombre. Los atributos traen la eficacia declarativa a C#, pero son una forma de metadatos de código y no actúan por sí mismos.

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Tutorial: Actualización de interfaces con métodos de interfaz predeterminados

Artículo • 22/03/2023

Puede definir una implementación cuando se declara 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

Debe configurar la máquina para ejecutar .NET, incluido el compilador de C#. El compilador de C# está disponible con [Visual Studio 2022](#) o el [SDK de .NET](#).

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:

C#

```
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:

C#

```
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. ¡No hagas un cambio importante!" Debe usar implementaciones de interfaz predeterminadas para actualizar 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. Estas características lo convierten 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:

C#

```
// 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:

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5,  
31))  
{  
    Reminders =  
    {  
        { new DateTime(2010, 08, 12), "child'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:

C#

```
// 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

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:

C#

```
// 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 otros campos 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:

C#

```
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:

C#

```
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":

C#

```
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 implementadas con varias clases. Esto facilita la actualización de esas definiciones de interfaz cuando se descubren nuevos requisitos para esa misma idea funcional.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

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

Artículo • 31/07/2024

Puede definir una implementación cuando se declara 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 a:

- ✓ 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

Debe configurar la máquina para ejecutar .NET, incluido el compilador de C#. El compilador de C# está disponible con [Visual Studio 2022](#) o el [SDK de .NET](#).

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.

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 proporcionan 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:

C#

```
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:

C#

```
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:

C#

```
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`:

C#

```
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.");
    }
}
```

La clase `OverheadLight` puede implementar la función de temporizador mediante la declaración de la compatibilidad con la interfaz:

C#

```
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:

C#

```
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:

C#

```
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:

C#

```
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`:

C#

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

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

C#

```
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:

C#

```
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:

C#

```
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:

C#

```
public enum PowerStatus
{
    NoPower,
```

```
    ACPower,  
    FullBattery,  
    MidBattery,  
    LowBattery  
}
```

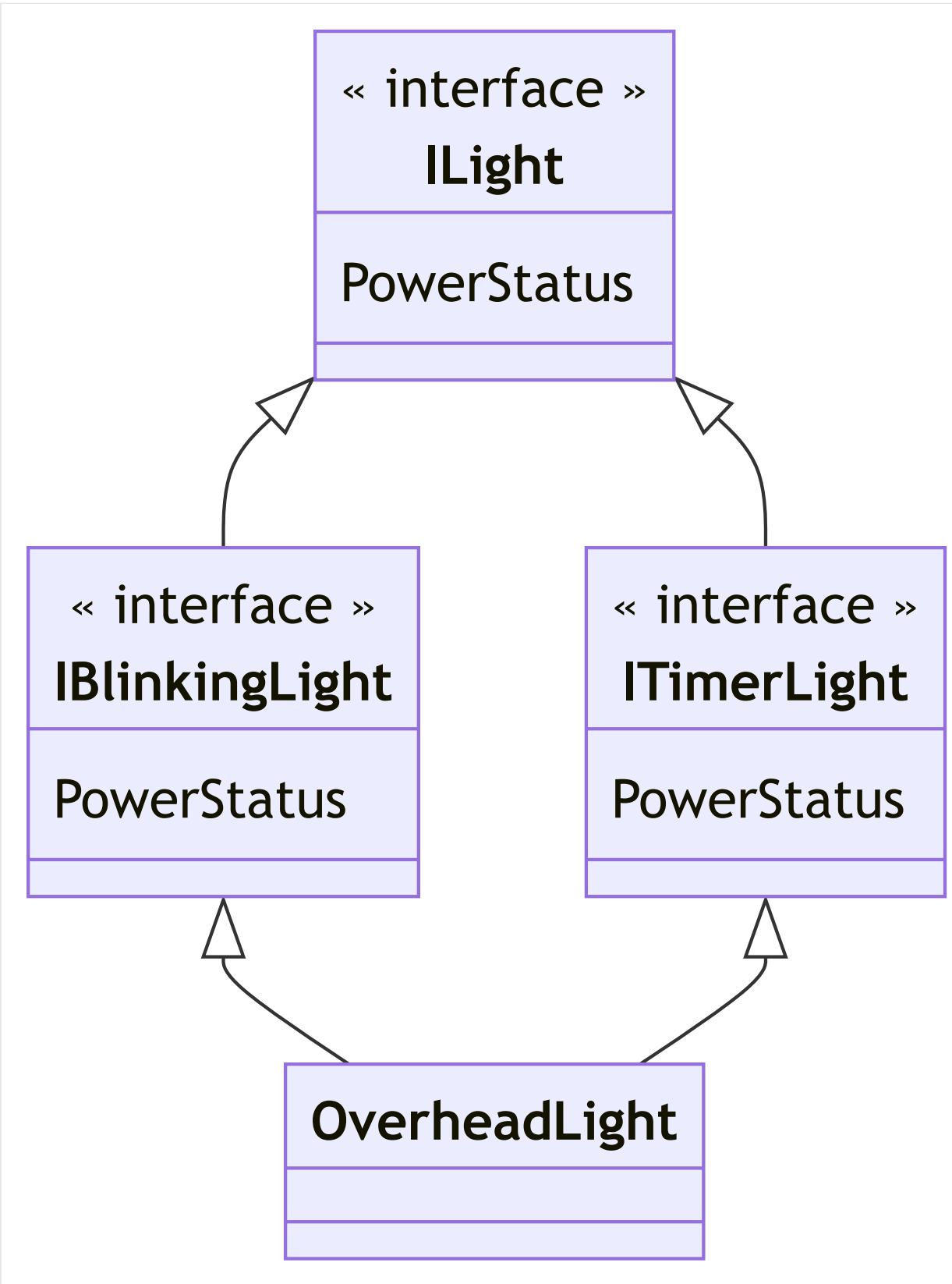
La implementación predeterminada presupone la ausencia de alimentación:

C#

```
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` implementaran 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. Esta situación se muestra en el diagrama siguiente:



En el diagrama anterior se muestra la ambigüedad. **OverheadLight** no proporciona una implementación de **ILight.PowerStatus**. Tanto **IBlinkingLight** como **ITimerLight** proporcionan invalidaciones más específicas. Una llamada a **ILight.PowerStatus** en una instancia de **OverheadLight** es ambigua. Debe agregar una nueva invalidación en **OverheadLight** para resolver la ambigüedad.

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.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Expression Trees

Article • 05/29/2024

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read [the LINQ tutorial](#) and the article about [lambda expressions](#) before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You already write code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables [Entity Framework](#) to translate the query you wrote in C# into SQL that executes in the database engine. Another example is [Moq](#), which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to use expression trees to build dynamic queries](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (CIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the `System.Linq.Expressions` namespace.

When a lambda expression is assigned to a variable of type `Expression<TDelegate>`, the compiler emits code to build an expression tree that represents the lambda expression.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on [building expression trees](#).

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on [translating expression trees](#).

Once you build an expression tree, you [execute the code represented by the expression tree](#).

Limitations

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods removed from the output
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (=) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- Collection expressions
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments, or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator, or the pattern matching switch expression
- COM call with ref omitted on the arguments
- ref, in or out parameters, ref return values, out arguments, or any values of ref struct type

Árboles de expresión: datos que definen el código

Artículo • 13/03/2023

Los árboles de expresiones son estructuras de datos que definen código. Los árboles de expresión se basan en las mismas estructuras que usa un compilador para analizar el código y generar el resultado compilado. A medida que lea este artículo, 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 sugieren 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 diferente a las API de Roslyn. Aquí tiene una línea de código:

C#

```
var sum = 1 + 2;
```

Si analiza el código anterior 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`, 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. En este caso, ambos nodos son expresiones constantes: el operando izquierdo es el valor `1` y el operando derecho es 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`)

El árbol anterior puede parecer complicado, pero es muy versátil. Siguiendo el mismo proceso, descompone expresiones mucho más complicadas. Tomemos esta expresión como ejemplo:

C#

```
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 va 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 navega con tanta facilidad como en el primer ejemplo. Siga recorriendo los nodos secundarios para buscar nodos hoja en la expresión. Los nodos primarios tienen 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, puede 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# analiza los programas de C# más complejos y crea 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, los árboles de expresiones facilitan la escritura de algoritmos que inspeccionan el código antes de ejecutarlo. Escriba un método cuyos argumentos sean expresiones y, a continuación, examine esas expresiones antes de ejecutar el código. El árbol de expresión es una representación completa del código: se ven los valores de cualquier subexpresión. Ve los nombres de propiedad y método. Ve el valor de las expresiones constantes. Convierte 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.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Compatibilidad del entorno de ejecución de .NET con árboles de expresión

Artículo • 13/03/2023

Hay una amplia lista de clases en el entorno de ejecución de .NET que funcionan con árboles de expresiones. En [System.Linq.Expressions](#) puede ver la lista completa. En lugar de enumerar la lista completa, vamos a explicar cómo se han diseñado las clases del entorno de ejecución.

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).

System.Linq.Expression y tipos derivados

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. Esta 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. 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, lo convierte en ese tipo y realiza 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 imprime el nombre de una variable para una expresión de acceso a la variable. El siguiente código muestra la práctica de comprobar el tipo de nodo, convertirlo en una expresión de acceso a la variable y después comprobar las propiedades del tipo de expresión específico:

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0]; -- first

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Creación de árboles de expresión

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:

C#

```
// 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#.

Navegación 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 se usan para trabajar con árboles de expresiones:

1. Fíjese en los miembros de la enumeración `ExpressionType` para determinar los posibles nodos que debe examinar. Esta lista ayuda 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.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Ejecución de árboles de expresión

Artículo • 13/03/2023

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 representado mediante un árbol de expresión, debe convertirlo en instrucciones de lenguaje intermedio ejecutables. 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.

ⓘ Nota

Si el tipo del delegado es desconocido, es decir, la expresión lambda es de tipo [LambdaExpression](#) y no [Expression<TDelegate>](#), llame 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.

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 en lo que significaría ejecutar directamente un elemento

[System.Linq.Expressions.ConstantExpression](#). ¿Significaría algo útil?) Cualquier árbol de expresión que sea un elemento [System.Linq.Expressions.LambdaExpression](#) o un tipo derivado de [LambdaExpression](#) se puede convertir en IL. El tipo de expresión [System.Linq.Expressions.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. 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`. Tiene 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.

En la mayoría de los casos, existe 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 `System.Linq.Expressions.LambdaExpression` contiene los miembros `LambdaExpression.Compile` y `LambdaExpression.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 `System.Reflection.Emit.MethodBuilder` con el lenguaje intermedio que representa la salida compilada del árbol de expresión.

ⓘ Importante

`CompileToMethod` solo está disponible en .NET Framework, no en .NET Core o .NET 5 y versiones posteriores.

Como opción, también puede proporcionar un `System.Runtime.CompilerServices.DebugInfoGenerator` que recibe la información de depuración de símbolos para el objeto de delegado generado. `DebugInfoGenerator` proporciona información de depuración completa sobre el delegado generado.

Para convertir una expresión en un delegado se usaría el siguiente código:

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

En el ejemplo de código siguiente se muestran los tipos concretos que se usan al compilar y ejecutar un árbol de expresión.

C#

```
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.
```

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.

C#

```
// 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
```

Ejecución y duraciones

El código se ejecuta mediante la invocación del delegado que se crea al llamar a `LambdaExpression.Compile()`. El código anterior, `add.Compile()`, devuelve un delegado. Para invocar ese delegado, llame a `func()`, que 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 crea un delegado que ejecuta el mismo código).

⊗ Precaución

No cree ningún 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 es una operación que consume mucho tiempo de ejecución. Probablemente 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 si 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. El compilador garantiza que las variables estén en el ámbito. 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`:

C#

```

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 la siguiente clase (bastante artificiosa) que implementa `System.IDisposable`:

C#

```

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 en el siguiente código, obtiene una `System.ObjectDisposedException` al ejecutar el código al que hace referencia la propiedad `Resource.Argument`:

C#

```

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 desde este método, se produce 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 entra cuando trabaja con árboles de expresión.

Hay numerosas 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 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 produce una excepción `ReferencedAssemblyNotFoundException`.

Resumen

Los árboles de expresión que representan expresiones lambda se pueden compilar para crear un delegado que se puede ejecutar. Los árboles de expresión proporcionan 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 predecibles y se detectan en las primeras pruebas de cualquier código que use los árboles de expresión.

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Interpretación de expresiones

Artículo • 13/03/2023

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.

C#

```
// 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
```

Ahora, vamos a escribir código para examinar la estructura de un *árbol de expresión*. Cada nodo de un árbol de expresión es 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.

Examen de 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:

C#

```
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}");
```

El código anterior imprime la siguiente salida:

Resultados

```
This is a/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.

Expresión de suma

Comencemos con el ejemplo de adición de la instrucción de esta sección.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

ⓘ Nota

No use `var` para declarar este árbol de expresión, porque el tipo natural del delegado es `Func<int>`, no `Expression<Func<int>>`.

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`. Hace 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:

C#

```

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:

Resultados

```

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 mucha repetición 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:

```
C#  
  
using System.Linq.Expressions;  
  
namespace Visitors;  
// 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 => node.NodeType;  
    public static Visitor CreateFromExpression(Expression node) =>  
        node.NodeType switch  
        {  
            ExpressionType.Constant => new  
ConstantVisitor((ConstantExpression)node),  
            ExpressionType.Lambda => new  
LambdaVisitor((LambdaExpression)node),  
            ExpressionType.Parameter => new  
ParameterVisitor((ParameterExpression)node),  
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),  
            _ => throw new NotImplementedException($"Node not processed yet:  
{node.NodeType}"),  
        };  
}  
  
// 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($"{prefix}The return type is {node.ReturnType}");  
        Console.WriteLine($"{prefix}The expression has  
{node.Parameters.Count} argument(s). They are:");  
        // Visit each parameter:  
        foreach (var argumentExpression in node.Parameters)  
        {  
            var argumentVisitor = CreateFromExpression(argumentExpression);  
            argumentVisitor.Visit(prefix + "\t");  
        }  
    }  
}
```

```

        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = 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($"{prefix}This binary expression is a {NodeType} expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = 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}, 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 visita cualquier `LambdaExpression` arbitrario. El código que creó solo busca una muestra 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 en la consola de error cuando se encuentra un tipo de nodo nuevo. De este modo, sabe que se va a agregar un tipo de expresión nuevo).

Cuando ejecuta este visitante en la expresión anterior, obtiene la siguiente salida:

Resultados

```
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.

Expresión de suma con más operandos

Vamos a probar un ejemplo más complicado, todavía limitando los tipos de nodo solo a los de adición:

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Antes de que ejecute estos ejemplos 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:

C#

```
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, realiza 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:

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Cree un visitante para esta suma y ejecute el visitante para ver esta salida:

Resultados

```
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
```

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):

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Aquí se muestra el resultado del visitante:

Resultados

```
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 de la salida. 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.

Ampliación 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. Vamos a hacer que funcione para la siguiente expresión factorial:

C#

```

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 ha 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 puede usar bucles, bloques, instrucciones IF/ELSE ni otras estructuras de control comunes en C#. Estoy limitado al uso de expresiones. En segundo lugar, no puede llamar recursivamente a la misma expresión. Podría si ya fuera un delegado, pero no puede llamarla en su forma de árbol de expresión. En la sección sobre la [creación de árboles de expresión](#), obtiene las técnicas para superar estas limitaciones.

En esta expresión, encuentra 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, ha visto cada uno de los nodos potenciales. Después, tiene todo lo que necesita. El resultado será similar al siguiente:

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant    => new
        ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda      => new
        LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter   => new
        ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add         => new
        BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal       => new
        BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply    => new BinaryVisitor((BinaryExpression)
        node),
        ExpressionType.Conditional => new
        ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call        => new
        MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
        {node.NodeType}"),
    };
}
```

`ConditionalVisitor` y `MethodCallVisitor` procesan esos dos nodos:

C#

```
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:

Resultados

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:

ByRef: False

This is an Parameter expression type
Type: System.Int32, Name: factor,

Ampliación de la biblioteca de ejemplo

Los ejemplos de esta sección muestran las técnicas principales para visitar y examinar nodos de un árbol de expresión. Ha simplificado los tipos de nodos que encontrará para concentrarse en las tareas principales de visitar y acceder a los nodos en un árbol de expresiones.

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 usada en este artículo se ha creado con fines de demostración y aprendizaje. No está optimizada. Aclara las estructuras y resalta las técnicas usadas para visitar los nodos y analizar lo que hay ahí.

Incluso con esas limitaciones, se encuentra en el camino correcto para escribir algoritmos que lean y comprendan árboles de expresión.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Creación de árboles de expresión

Artículo • 13/03/2023

El compilador de C# creó todos los árboles de expresión que ha visto hasta ahora. Ha creado una expresión lambda asignaba a una variable de tipo `Expression<Func<T>>` o de algún tipo similar. En muchos escenarios, crea una expresión en memoria en tiempo de ejecución.

Los árboles de expresión son inmutables. Inmutable significa que debe crear el árbol desde las hojas hasta la raíz. Las API que usa para crear los árboles de expresión reflejan este hecho: los métodos que usa para crear un nodo toman todos sus elementos secundarios como argumentos. Veamos algunos ejemplos para mostrarle las técnicas.

Creación de nodos

Empezaremos con la expresión de adición con la que ha estado trabajando en estas secciones:

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

Para crear ese árbol de expresión, primero cree los nodos hoja. Los nodos hoja son constantes. Use el método `Constant` para crear los nodos:

C#

```
var one = Expression.Constant(1, typeof(int));  
var two = Expression.Constant(2, typeof(int));
```

Después, cree la expresión de adición:

C#

```
var addition = Expression.Add(one, two);
```

Una vez que haya creado la expresión de adición, se crea la expresión lambda:

C#

```
var lambda = Expression.Lambda(addition);
```

Esta expresión lambda no contiene argumentos. Posteriormente en esta sección, verá cómo asignar argumentos a parámetros y crear expresiones más complicadas.

Para las expresiones como esta, puede combinar todas las llamadas en una sola instrucción:

```
C#  
  
var lambda2 = Expression.Lambda(  
    Expression.Add(  
        Expression.Constant(1, typeof(int)),  
        Expression.Constant(2, typeof(int))  
    )  
);
```

Creación de un árbol

En la sección anterior se muestran los conceptos básicos de la creación de un árbol de expresión en 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 crea 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:

```
C#  
  
Expression<Func<double, double, double>> distanceCalc =  
    (x, y) => Math.Sqrt(x * x + y * y);
```

Comienza creando expresiones de parámetro para `x` y `y`:

```
C#  
  
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:

```
C#  
  
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`.

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ??
    throw new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

La llamada `GetMethod` podría devolver `null` si no se encuentra el método. Lo más probable es que se deba a que ha escrito mal el nombre del método. De otro modo, podría significar que el ensamblado necesario no se carga. Por último, coloque la llamada al método en una expresión lambda y asegúrese de definir los argumentos en dicha expresión:

C#

```
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 `System.Reflection.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 extienden a otros árboles de expresión.

Compilación de 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:

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

El código anterior no ha compilado el árbol de expresiones, 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. 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.

C#

```
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 le gustaría evitar en nuestras tareas de codificación diarias.

En esta sección, ha escrito código 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. Pruebelo compilando y ejecutando los ejemplos.

Asignación de construcciones de código a expresiones

En el siguiente ejemplo de código se muestra un árbol de expresión que represente la expresión lambda `num => num < 5` mediante la API.

C#

```
// 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 });
```

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.

C#

```
// 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.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Traslado de árboles de expresión

Artículo • 13/03/2023

En este artículo, obtendrá información sobre cómo visitar cada nodo en un árbol de expresión, mientras se crea una copia modificada de ese árbol de expresión. Trasladará los árboles de expresión para comprender los algoritmos para poder trasladarlos a otro entorno. Cambiará el algoritmo que se ha creado. Podría agregar el registro, interceptar las llamadas de método y realizar un seguimiento de ellas, o con otros fines.

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 haya colocado en el árbol.

Visitaremos un árbol de expresión y crearemos 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, dejará el árbol de expresión intacto. En lugar de leer el valor de la constante y reemplazarlo con una constante nueva, hará este cambio 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:

C#

```
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;
}
```

Cree un nuevo árbol reemplazando el nodo original por el sustituto. Puede comprobar los cambios mediante la compilación y ejecución del árbol reemplazado.

C#

```
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 el ejemplo anterior 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. Los nodos se pueden usar en ambos árboles porque los nodos del árbol existente no se pueden modificar. La reutilización de nodos da lugar a importantes eficiencias de memoria. 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 comprobar el nuevo árbol mediante la creación de un segundo visitante que recorre el árbol de nodos de adición y calcula el resultado. Haga 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.

C#

```
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 you 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 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, dichos elementos han 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:

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"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 expresión `sum`, produce el siguiente resultado:

Resultados

```
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`:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Este es el resultado de examinar esta expresión:

Resultados

```
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 diferente. Los nodos se recorren en un orden diferente, porque el árbol se construyó con diferentes operaciones que se producen en primer lugar.

Creación de una copia modificada

Cree un nuevo proyecto de **aplicación de consola**. Agregue una directiva `using` al archivo para el espacio de nombres `System.Linq.Expressions`. Agregue la clase `AndAlsoModifier` al proyecto.

C#

```
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);
            return Expression.OrElse(left, right);
        }
        else
            return base.VisitBinary(b);
    }
}
```

```

        // 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. 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 clase 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.

Agregue una directiva `using` al archivo para el espacio de nombres `System.Linq.Expressions`. Agregue código al método `Main` en el archivo `Program.cs` para crear un árbol de expresión y pasarlo al método que lo modifica.

C#

```

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. Compile y ejecute la aplicación.

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 información sobre la compilación de 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.

Ahora ha visto la verdadera eficacia de los árboles de expresión. Examine un conjunto de código, realice los cambios que quiera en ese código y ejecute la versión modificada. Como los árboles de expresión son inmutables, crea árboles nuevos mediante el uso de los componentes de árboles existentes. La reutilización de los nodos minimiza la cantidad de memoria necesaria para crear árboles de expresión modificados.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

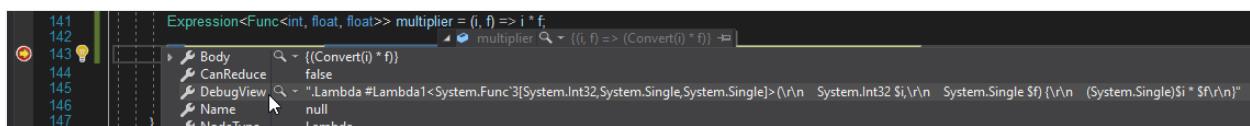
 [Open a documentation issue](#)

 [Provide product feedback](#)

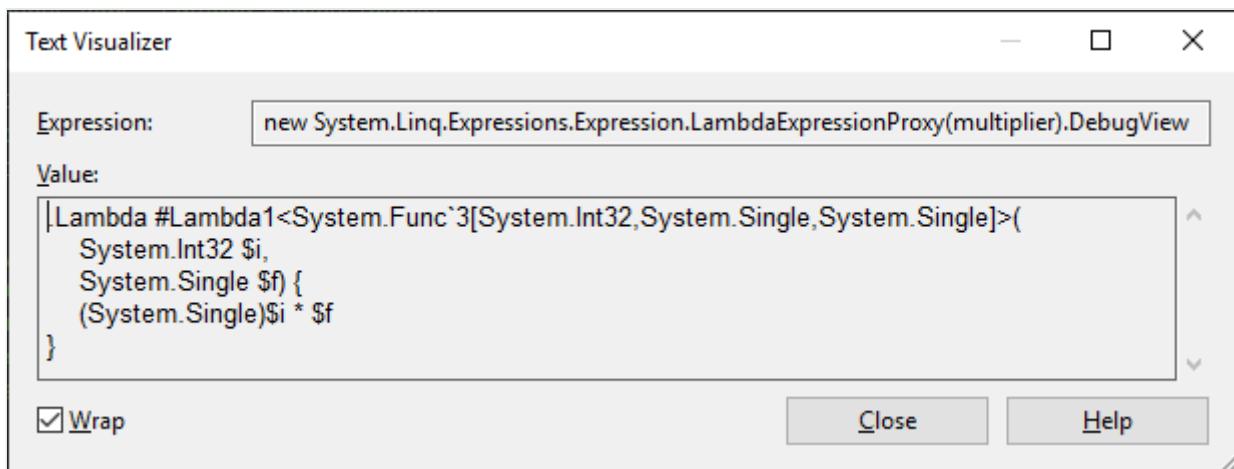
Depuración de árboles de expresión en Visual Studio

Artículo • 13/03/2023

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](#). `DebugView` solo está disponible en modo de depuración.

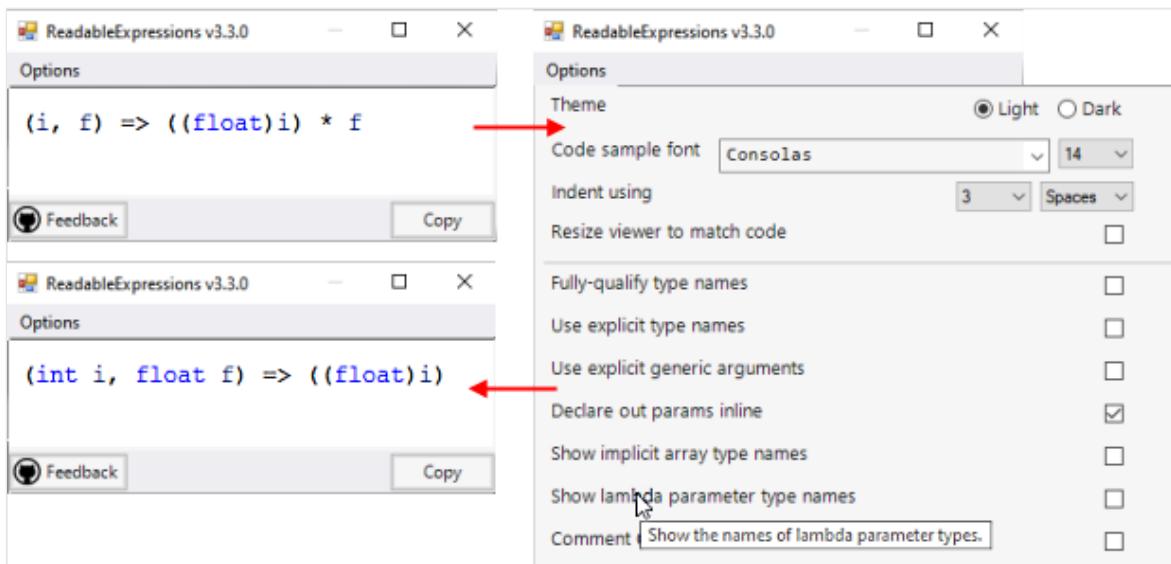


Puesto que `DebugView` es una cadena, puede usar el [visualizador de texto integrado](#) para verla en varias líneas si selecciona **Visualizador de texto** en el ícono de lupa situado junto a la etiqueta `DebugView`.

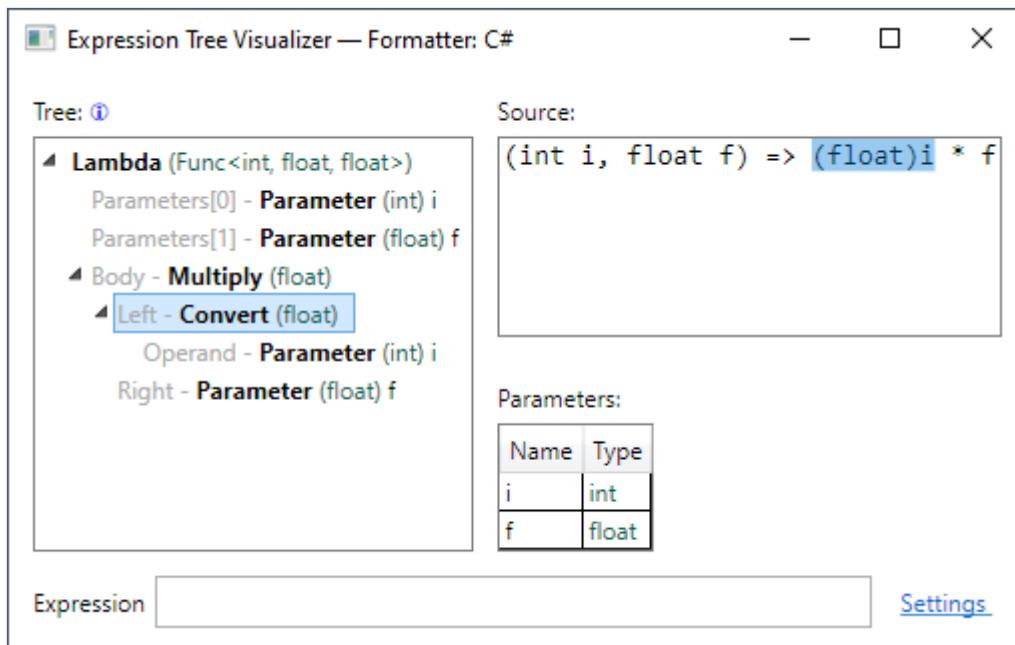


Como alternativa, puede instalar y usar [un visualizador personalizado](#) para árboles de expresión, como:

- [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:

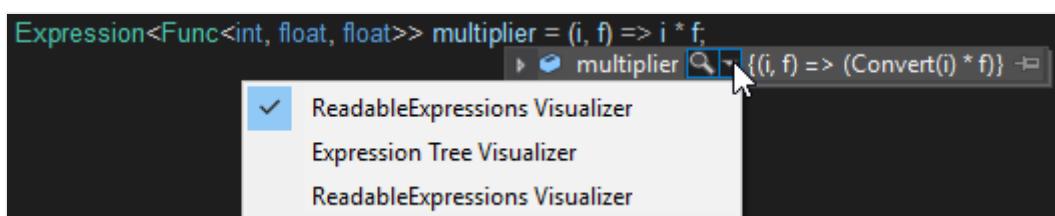


- [Expression Tree Visualizer](#) (licencia MIT), proporciona una vista de árbol del árbol de expresión y sus nodos individuales:



Apertura de un visualizador para un árbol de expresión

Seleccione 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:



Seleccione el visualizador que deseé usar.

Vea también

- [Depurar en Visual Studio](#)
- [Create Custom Visualizers](#) (Crear visualizadores personalizados)
- [Sintaxis DebugView](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Sintaxis de DebugView

Artículo • 13/03/2023

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 **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`.

```
C#  
  
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");  
/*  
 $num  
 */  
  
ParameterExpression numParam = Expression.Parameter(typeof(int));  
/*  
 $var1  
 */
```

ConstantExpression

Para los objetos **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
<code>System.UInt32</code>	<code>uint</code>	<code>U</code>

Tipo	Palabra clave	Sufijo
System.Int64	long	L
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

C#

```
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 [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.

C#

```
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 [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`.

C#

```
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 [LabelExpression](#), este valor se muestra antes del objeto [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`.

C#

```
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 `#+`.

C#

```
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  
*/
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Métodos

System.Linq.Expressions.Expression.Add

Artículo • 31/01/2024

En este artículo se proporcionan comentarios adicionales a la documentación de referencia de esta API.

El [Add](#) método devuelve un [BinaryExpression](#) que tiene la [Method](#) propiedad establecida en el método de implementación. La [Type](#) propiedad se establece en el tipo del nodo. Si se levanta el nodo, las [IsLifted](#) propiedades y [IsLiftedToNull](#) son `true`. De lo contrario, son `false`. La propiedad [Conversion](#) es `null`.

En la siguiente información se describe el método de implementación, el tipo de nodo y si se levanta un nodo.

Método de implementación

Las reglas siguientes determinan el método de implementación seleccionado para la operación:

- Si la [Type](#) propiedad de o `left right` representa un tipo definido por el usuario que sobrecarga el operador de suma, el [MethodInfo](#) que representa ese método es el método de implementación.
- De lo contrario, si `left` es `.` Escriba y `right`. El tipo son tipos numéricos, el método de implementación es `null`.

Tipo de nodo y elevado frente a no elevado

Si el método de implementación no `null` es :

- Si `left` es `.` Escriba y `right`. El tipo se puede asignar a los tipos de argumento correspondientes del método de implementación, el nodo no se levanta. El tipo del nodo es el tipo de valor devuelto del método de implementación.
- Si se cumplen las dos condiciones siguientes, el nodo se levanta y el tipo del nodo es el tipo que acepta valores NULL que corresponde al tipo devuelto del método de implementación:
 - `left`. Escriba y `right`. El tipo son tipos de valor de los que al menos uno acepta valores NULL y los tipos que no aceptan valores NULL correspondientes son

iguales a los tipos de argumento correspondientes del método de implementación.

- El tipo de valor devuelto del método de implementación es un tipo de valor que no acepta valores NULL.

Si el método de implementación es `null`:

- Si `left` es `.`. Escriba y `right`. El tipo no acepta valores NULL, el nodo no se levanta. El tipo del nodo es el tipo de resultado del operador de suma predefinido.
- Si `left` es `.`. Escriba y `right`. El tipo admite valores NULL y se levanta el nodo. El tipo del nodo es el tipo que acepta valores NULL que corresponde al tipo de resultado del operador de suma predefinido.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Clase System.Linq.Expressions.BinaryExpression

Artículo • 11/01/2024

En este artículo se proporcionan comentarios adicionales a la documentación de referencia de esta API.

La [BinaryExpression](#) clase representa una expresión que tiene un operador binario.

En las tablas siguientes se resumen los métodos de fábrica que se pueden usar para crear un [BinaryExpression](#) que tenga un tipo de nodo específico, representado por la [NodeType](#) propiedad. Cada tabla contiene información para una clase específica de operaciones como aritmética o bit a bit.

Operaciones aritméticas binarias

 Expandir tabla

Tipo de nodo	Factory (método)
Add	Add
AddChecked	AddChecked
Divide	Divide
Modulo	Modulo
Multiply	Multiply
MultiplyChecked	MultiplyChecked
Power	Power
Subtract	Subtract
SubtractChecked	SubtractChecked

Operaciones bit a bit

 Expandir tabla

Tipo de nodo	Factory (método)
And	And
Or	Or
ExclusiveOr	ExclusiveOr

Operaciones de desplazamiento

[+] Expandir tabla

Tipo de nodo	Factory (método)
LeftShift	LeftShift
RightShift	RightShift

Operaciones booleanas condicionales

[+] Expandir tabla

Tipo de nodo	Factory (método)
AndAlso	AndAlso
OrElse	OrElse

Operaciones de comparación

[+] Expandir tabla

Tipo de nodo	Factory (método)
Equal	Equal
NotEqual	NotEqual
GreaterThanOrEqual	GreaterThanOrEqual
GreaterThan	GreaterThan
LessThan	LessThan

Tipo de nodo	Factory (método)
LessThanOrEqual	LessThanOrEqual

Operaciones de fusión

 [Expandir tabla](#)

Tipo de nodo	Factory (método)
Coalesce	Coalesce

Operaciones de indexación de matrices

 [Expandir tabla](#)

Tipo de nodo	Factory (método)
ArrayIndex	ArrayIndex

Además, los [MakeBinary](#) métodos también se pueden usar para crear un [BinaryExpression](#). Estos métodos de fábrica se pueden usar para crear un [BinaryExpression](#) de cualquier tipo de nodo que represente una operación binaria. El parámetro de estos métodos que es de tipo [NodeType](#) especifica el tipo de nodo deseado.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see [Consuming Unmanaged DLL Functions](#) and [How to use platform invoke to play a WAV file](#).

ⓘ Note

The [Common Language Runtime](#) (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see [Security in .NET](#).

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the `/clr` compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use `regsvr32.exe` to register or un-register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the [Tlbimp.exe \(Type Library Importer\)](#), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see [Exposing COM Components to the .NET Framework](#).

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see [Assembly Information Dialog Box](#).
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the [Regasm.exe \(Assembly Registration Tool\)](#), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see [Exposing .NET Framework Components to COM](#) and [Example COM Class](#).

See also

- [Improving Interop Performance](#)
- [Introduction to Interoperability between COM and .NET](#)
- [Introduction to COM Interop in Visual Basic](#)
- [Marshaling between Managed and Unmanaged Code](#)
- [Interoperating with Unmanaged Code](#)
- [Registering Assemblies with COM](#)

Clase COM de ejemplo

Artículo • 10/05/2023

El código siguiente es un ejemplo de una clase que se expondría como un objeto COM. Una vez que coloque este código en un archivo .cs 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 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 declare en estas interfaces no serán visibles para COM, pero lo serán para otros 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 que se usa 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 es la interfaz de clase predeterminada. Implemente los métodos y propiedades expuestos para COM aquí. Deben ser públicos y coincidir con las declaraciones de la interfaz de clase. Asimismo, declare los eventos iniciados por la clase aquí. Deben ser públicos y coincidir con las declaraciones de la interfaz de eventos.

Ejemplo

C#

```
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
{
}
}
```

Consulte también

- Interoperabilidad
- Página Compilar (Diseñador de proyectos) (C#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Programación de Office en C#

Artículo • 09/03/2023

C# ofrece características 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.

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: Insertar 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.

Importante

VSTO (Visual Studio Tools para Office) se basa en [.NET Framework](#). Los complementos COM también se pueden escribir con .NET Framework. No se pueden crear complementos de Office con [.NET Core y .NET 5 o versiones posteriores](#), las últimas versiones de .NET. Esto se debe a que .NET Core/.NET 5 o versiones posteriores no pueden funcionar junto con .NET Framework en el mismo proceso, y se pueden provocar errores de carga de complementos. Puede seguir usando .NET Framework a fin de escribir complementos VSTO y COM para Office. Microsoft no actualizará VSTO ni la plataforma de complementos COM para usar .NET Core, o .NET 5 o versiones posteriores. Puede aprovechar .NET Core y .NET 5 o versiones posteriores, incluido ASP.NET Core, para crear el lado servidor de [complementos web de Office](#).

Requisitos previos

Debe tener Microsoft Office Excel y Microsoft Office Word instalados en su equipo para completar este tutorial.

Nota

Es posible que tu 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](#).

Configuración de una aplicación complemento de Excel

1. Inicie Visual Studio.
2. En el menú **Archivo**, seleccione **Nuevo** y haga clic en **Proyecto**.
3. En el panel **Plantillas instaladas**, expanda **C#, Office** y, después, seleccione el año de versión del producto de Office.
4. En el panel **Plantillas**, seleccione **Excel <versión> 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. Seleccione **Aceptar**.
8. El proyecto nuevo aparece en el **Explorador de soluciones**.

Agregar referencias

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto y luego seleccione **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, es posible que tenga que instalarlos (vea [Procedimientos para instalar ensamblados de interoperabilidad primarios de Office](#)).
3. Seleccione **Aceptar**.

Incorporación de las instrucciones Imports necesarias o las directivas using

En el **Explorador de soluciones**, haga clic con el botón derecho en el archivo `ThisAddIn.cs` y luego seleccione **Ver código**. Agregue las directivas `using` (C#) siguientes en la parte superior del archivo de código si no están ya presentes.

```
C#
```

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Creación de una lista de cuentas bancarias

En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto, seleccione **Agregar** y luego **Clase**. Asigne un nombre a la clase `Account.cs`. Seleccione **Agregar**. Reemplace la definición de la clase `Account` por el código siguiente. Las definiciones de clase usan *propiedades implementadas automáticamente*.

```
C#
```

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Para crear una lista `bankAccounts` que contenga dos cuentas, agregue el código siguiente al método `ThisAddIn_Startup` en `ThisAddIn.cs`. Las declaraciones de lista usan *inicializadores de colección*.

```
C#
```

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
}
```

```
    }  
};
```

Exportación de datos a Excel

En el mismo archivo, agregue el siguiente método a la clase `ThisAddIn`. El método configura un libro de Excel, a donde exporta los datos.

C#

```
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();  
}
```

- El método `Add` tiene un *parámetro opcional* para especificar una plantilla determinada. Los parámetros opcionales permiten omitir el argumento de ese parámetro si se desea utilizar el valor predeterminado del parámetro. Como el ejemplo anterior no tiene argumentos, `Add` usa la plantilla predeterminada y crea un libro. 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 [Argumentosopcionalesyconnombre](#).
- 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.

C#

```
// Visual C# 2010 provides indexed properties for COM programming.  
excelApp.Range["A1"].Value = "ID";  
excelApp.ActiveCell.Offset[1, 0].Select();
```

No se pueden crear propiedades indexadas propias. La característica solo admite el uso de las propiedades indizadas existentes.

Agregue el código siguiente al final de `DisplayInExcel` para ajustar los anchos de columna a fin de adaptarlos al contenido.

C#

```
excelApp.Columns[1].AutoFit();  
excelApp.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`. Los objetos COM se tratan como `dynamic` de forma automática cuando **Incrustar tipos de interoperabilidad** tiene su valor predeterminado, `True`, o equivalentemente, al hacer referencia al ensamblado con la opción del compilador `EmbedInteropTypes`. Para obtener más información sobre cómo insertar tipos de interoperabilidad, consulte los procedimientos "Búsqueda de la referencia a un PIA" y "Restauración de la dependencia de un PIA" más adelante en este artículo. Para obtener más información sobre `dynamic`, vea [dynamic](#) o [Uso de tipo dinámico](#).

Invocación de `DisplayInExcel`

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 procesadas. El segundo argumento es una expresión lambda de varias líneas que define cómo procesar 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](#).

C#

```
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;
    }
});
```

Presione F5 para ejecutar el programa. Aparece una hoja de cálculo de Excel que contiene los datos de las cuentas.

Incorporación de un documento de Word

Agregue el código siguiente al final del método `ThisAddIn_StartUp` para crear un documento de Word que contenga un vínculo al libro de Excel.

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

En este código se muestran varias de las características de C#: la capacidad de omitir la palabra clave `ref` en programación COM, argumentos con nombre y argumentos opcionales. El método `PasteSpecial` tiene siete parámetros y todos ellos son parámetros de referenciaopcionales. 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, los argumentos indican la creación de un vínculo al libro en el Portapapeles (parámetro `Link`) y que el vínculo en el documento de Word se muestra como un ícono (parámetro `DisplayAsIcon`). C# también le permite omitir la palabra clave `ref` para estos argumentos.

Ejecución de la aplicación

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.

Limpieza del proyecto completado

En Visual Studio, seleccione **Limpiar solución** en el menú **Compilación**. De lo contrario, el complemento se ejecutará cada vez que abra Excel en el equipo.

Búsqueda de la referencia de PIA

1. Vuelva a ejecutar la aplicación, pero no seleccione **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. El ensamblado incluye los espacios de nombres `Microsoft.Office.Interop.Excel` y `Microsoft.Office.Interop.Word`. 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: Consulta del contenido de un ensamblado](#).
5. Haga doble clic en el ícono **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 en la lista. Dado que importó los tipos que necesita el proyecto en el ensamblado, no es necesario instalar referencias a un PIA. La importación de los tipos en el ensamblado facilita la implementación. Los PIA no tienen que estar presentes en el equipo del usuario. Una aplicación no requiere la implementación de una versión específica de un PIA. Las aplicaciones pueden funcionar con varias versiones de Office, siempre que existan las API necesarias 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. El código no puede usar ninguna API que no esté disponible en la versión de Office con la que trabaja. No siempre está claro si una API determinada estaba disponible en una versión anterior. No se recomienda trabajar con versiones anteriores de Office.
6. Cierre la ventana del manifiesto y la del ensamblado.

Restauración de la dependencia de PIA

1. En el Explorador de soluciones, seleccione 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 ícono 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, seleccione Limpiar solución en el menú Compilación para limpiar el proyecto completado.

Consulte también

- [Propiedades autoimplementadas \(C#\)](#)
- [Inicializadores de objeto y colección](#)
- [Visual Studio Tools para Office \(VSTO\)](#)
- [Argumentos opcionales y con nombre](#)
- [dynamic](#)
- [Uso de tipo dinámico](#)
- [Expresiones lambda \(C#\)](#)
- [Tutorial: Inserción de información de tipos de los ensamblados de Microsoft Office en Visual Studio](#)
- [Tutorial: Inserción de tipos de ensamblados administrados](#)
- [Tutorial: Creación del primer complemento VSTO para Excel](#)



Collaborate with us on



.NET feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Procedimiento para usar la invocación de plataforma para reproducir un archivo WAV

Artículo • 07/04/2023

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 onda, 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, seleccione **Abrir** para reproducirlo mediante la invocación de plataforma. Un cuadro de texto muestra la ruta de acceso completa del archivo seleccionado.

```
C#  
  
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:\";
    //<Snippet5>
    dialog1.Filter = "Wav Files (*.wav)|*.wav";
    //</Snippet5>
    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.
}
}

```

El cuadro de diálogo **Abrir archivos** se puede filtrar con los valores correspondientes para mostrar solo los archivos que tengan la extensión .wav.

Compilación del código

Cree un proyecto de aplicación Windows Forms para C# en Visual Studio y asígnele el nombre **WinSound**. Copie el código anterior y péguelo sobre el contenido del archivo

Form1.cs. Copie el código siguiente y péguelo en el archivo *Form1.Designer.cs*, en el método `InitializeComponent()`, después de cualquier código existente.

C#

```
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();
```

Compile y ejecute el código.

Consulte también

- [Aproximación a la invocación de plataforma](#)
- [Serialización de datos con invocación de plataforma](#)



Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Procedimiento para usar propiedades indexadas en la programación de interoperabilidad COM

Artículo • 08/03/2023

Las propiedades indexadas funcionan junto con otras características de C#, como los [argumentos con nombre y optionales](#), un nuevo tipo ([dinámico](#)) y la [información de tipo incrustada](#), para mejorar la programación en Microsoft Office.

Importante

VSTO (Visual Studio Tools para Office) se basa en [.NET Framework](#). Los complementos COM también se pueden escribir con .NET Framework. No se pueden crear complementos de Office con [.NET Core y .NET 5 o versiones posteriores](#), las últimas versiones de .NET. Esto se debe a que .NET Core/.NET 5 o versiones posteriores no pueden funcionar junto con .NET Framework en el mismo proceso, y se pueden provocar errores de carga de complementos. Puede seguir usando .NET Framework a fin de escribir complementos VSTO y COM para Office. Microsoft no actualizará VSTO ni la plataforma de complementos COM para usar .NET Core, o .NET 5 o versiones posteriores. Puede utilizar .NET Core y .NET 5 o versiones posteriores, incluido ASP.NET Core, para crear el lado servidor de [complementos web de 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\[\]](#) 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 ejemplo siguiente.

```
{language}
```

```
// 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:

```
{language}
```

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.Range["A1"];
```

En el ejemplo anterior, también se usa la característica [argumentos opcionales](#), que le permite omitir `Type.Missing`.

Las propiedades indexadas permiten escribir el siguiente código.

```
{language}
```

```
// Visual C# 2010.  
targetRange.Value = "Name";
```

No se pueden crear propiedades indexadas 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# \(Guía de programación de C#\)](#).

```
{language}
```

```
// 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();  
        }  
  
        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](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cómo acceder a objetos de interoperabilidad de Office

Artículo • 09/03/2023

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 artículo se utilizarán las nuevas características para escribir código que crea y muestra una hoja de cálculo de Microsoft Office Excel. Se escribe código para agregar un documento de Office Word que contiene un ícono 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.

ⓘ Nota

Es posible que tu 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](#).

ⓘ Importante

VSTO (Visual Studio Tools para Office) se basa en [.NET Framework](#). Los complementos COM también se pueden escribir con .NET Framework. No se pueden crear complementos de Office con [.NET Core y .NET 5 o versiones posteriores](#), las últimas versiones de .NET. Esto se debe a que .NET Core/.NET 5 o versiones posteriores no pueden funcionar junto con .NET Framework en el mismo proceso, y se pueden provocar errores de carga de complementos. Puede seguir usando .NET Framework a fin de escribir complementos VSTO y COM para Office. Microsoft no actualizará VSTO ni la plataforma de complementos COM para usar .NET Core, o .NET 5 o versiones posteriores. Puede utilizar .NET Core y .NET 5 o versiones posteriores, incluido ASP.NET Core, para crear el lado servidor de [complementos web de Office](#).

Para crear una aplicación de consola nueva

1. Inicie Visual Studio.
2. En el menú **Archivo**, seleccione **Nuevo** y haga clic en **Proyecto**. Aparecerá el cuadro de diálogo **Nuevo proyecto**.
3. En el panel **Plantillas instaladas**, expanda **C#** y, a continuación, seleccione **Windows**.
4. En la parte superior del cuadro de diálogo **Nuevo proyecto**, asegúrese de seleccionar **.NET Framework 4** (o una versión posterior) como plataforma de destino.
5. En el panel **Plantillas**, seleccione **Aplicación de consola**.
6. Escriba un nombre para el proyecto en el campo **Nombre**.
7. Seleccione **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 seleccione **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, es posible que tenga que instalarlos. Vea [Cómo: Instalación de ensamblados de interoperabilidad primarios de Office](#).
3. Seleccione **Aceptar**.

Para agregar las directivas using necesarias

En el **Explorador de soluciones**, haga clic con el botón derecho en el archivo *Program.cs* y luego seleccione **Ver código**. Agregue las directivas `using` siguientes a la parte superior del archivo de código:

C#

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Para crear una lista de las cuentas bancarias

Pegue la definición de clase siguiente en `Program.cs`, bajo la clase `Program`.

C#

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Agregue el código siguiente al método `Main` para crear una lista `bankAccounts` lista que contenga dos cuentas.

C#

```
// 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 permiten omitir el argumento para ese parámetro si se desea utilizar el valor predeterminado del parámetro. Dado que no ha proporcionado ningún argumento, `Add` usa la plantilla predeterminada y crea un libro. La instrucción equivalente en versiones anteriores de C# requiere un argumento de marcador de posición:

```
ExcelApp.Workbooks.Add(Type.Missing).
```

C#

```
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 particular 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;
}
```

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.

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

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.

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

Agregue el código siguiente al final de `DisplayInExcel` para ajustar los anchos de columna a fin de adaptarlos al contenido.

C#

```
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.

```
C#
```

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

En C#, el valor devuelto `Object` se convierte automáticamente en `dynamic` si se hace referencia al ensamblado mediante la opción del compilador `EmbedInteropTypes` o, de forma equivalente, si la propiedad `Incrustar tipos de interoperabilidad` de Excel es true. El valor predeterminado de esta propiedad es true.

Para ejecutar el proyecto

Agregue la línea siguiente al final de `Main`.

```
C#
```

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

Presione CTRL+F5. Aparece una hoja de cálculo de Excel que contiene los datos de las dos cuentas.

Para agregar un documento de Word

El código siguiente abre una aplicación de Word y crea un ícono 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 características 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. Puede omitir los argumentos de cualquiera o de todos los parámetros si desea usar sus valores predeterminados.

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 ícono. Puede usar argumentos con nombre para esos dos argumentos y omitir los demás. Aunque estos argumentos son parámetros de referencia, no es necesario utilizar la palabra clave `ref` ni crear variables para enviarlas como argumentos. Puede enviar los valores directamente.

C#

```
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);
}
```

Agregue la siguiente instrucción al final de `Main`.

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

Agregue la siguiente instrucción al final de `DisplayInExcel`. El método `Copy` agrega la hoja de cálculo en el Portapapeles.

C#

```
// 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();
```

Presione CTRL+F5. Un documento de Word aparecerá con un ícono. Haga doble clic en el ícono para abrir la hoja de cálculo en primer plano.

Para establecer la propiedad **Incrustar tipos de interoperabilidad**

Es posible realizar más mejoras 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, la programación es más fácil porque el tipo `dynamic` representa los tipos requeridos y devueltos que se declaran en los métodos COM. 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](#).

El comportamiento predeterminado es insertar información de tipos en lugar de utilizar los ensamblados de interoperabilidad primarios. Debido a ese comportamiento predeterminado, algunos de los ejemplos anteriores se simplifican. No necesita ninguna 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 `Object` y `AutoFit` es un método de Excel. En el código siguiente se muestra la conversión.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

Para cambiar el valor predeterminado y usar los ensamblados de interoperabilidad primarios en lugar de incrustar información de tipos, expanda el nodo Referencias del Explorador de soluciones y, después, seleccione `Microsoft.Office.Interop.Excel` o `Microsoft.Office.Interop.Word`. Si no ve la ventana **Propiedades**, presione F4. Busque

Incrustar tipos de interoperabilidad en la lista de propiedades y cambie su valor a **False**. Del mismo modo, se puede compilar mediante la [opción del compilador References](#) en lugar de **EmbedInteropTypes** en un símbolo del sistema.

Para agregar formato adicional a la tabla

Reemplace las dos llamadas a **AutoFit** en **DisplayInExcel** con la siguiente instrucción.

C#

```
// 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 argumentosopcionales 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 correspondiente, no es necesario proporcionar el nombre de parámetro. Sin embargo, la instrucción puede ser más fácil de entender si se incluye el nombre del parámetro, como se muestra en el código siguiente.

C#

```
// Call to AutoFormat in Visual C# 2010.  
workSheet.Range["A1", "B3"].AutoFormat(Format:  
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

Presione CTRL+F5 para ver el resultado. Puede encontrar otros formatos incluidos en la enumeración [XlRangeAutoFormat](#).

Ejemplo

En el código siguiente se muestra el ejemplo completo.

C#

```
using System.Collections.Generic;  
using Excel = Microsoft.Office.Interop.Excel;  
using Word = Microsoft.Office.Interop.Word;  
  
namespace OfficeProgrammingWalkthruComplete  
{  
    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 particular  
        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; }
}

```

Consulte también

- [Type.Missing](#)

- dynamic
- Argumentos opcionales y con nombre
- Procedimiento para usar argumentos opcionales y con nombre en la programación de Office

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Uso de argumentosopcionales y con nombre en la programación de Office

Artículo • 08/03/2023

Los argumentos con nombre y los argumentos opcionales 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.

ⓘ Importante

VSTO (Visual Studio Tools para Office) se basa en [.NET Framework](#). Los complementos COM también se pueden escribir con .NET Framework. No se pueden crear complementos de Office con [.NET Core y .NET 5 o versiones posteriores](#), las últimas versiones de .NET. Esto se debe a que .NET Core/.NET 5 o versiones posteriores no pueden funcionar junto con .NET Framework en el mismo proceso, y se pueden provocar errores de carga de complementos. Puede seguir usando .NET Framework a fin de escribir complementos VSTO y COM para Office. Microsoft no actualizará VSTO ni la plataforma de complementos COM para usar .NET Core, o .NET 5 o versiones posteriores. Puede utilizar .NET Core y .NET 5 o versiones posteriores, incluido ASP.NET Core, para crear el lado servidor de [complementos web de Office](#).

En el ejemplo siguiente, el método [ConvertToTable](#) tiene 16 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 16 parámetros son opcionales, ya que la mayoría de las veces no interesa especificar valores concretos para todos ellos. Sin embargo, sin argumentos con nombre ni opcionales, debe proporcionar un valor o un valor de marcador de posición. Con los argumentos opcionales y con nombre, puede especificar valores solo para los parámetros necesarios para el proyecto.

Debe tener Microsoft Office Word instalado en el equipo para completar estos procedimientos.

ⓘ Nota

Es posible que tu 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 nueva aplicación de consola

Inicie Visual Studio. En el menú **Archivo**, seleccione **Nuevo** y haga clic en **Proyecto**. En el panel **Categorías de plantillas**, expanda **C#** y, a continuación, seleccione **Windows**. En la parte superior del panel **Plantillas**, asegúrese de que **.NET Framework 4** aparece en el cuadro **Plataforma de destino**. En el panel **Plantillas**, seleccione **Aplicación de consola**. Escriba un nombre para el proyecto en el campo **Nombre**. Seleccione **Aceptar**. El proyecto nuevo aparece en el **Explorador de soluciones**.

Agregar una referencia

En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto y luego seleccione **Agregar referencia**. Aparecerá el cuadro de diálogo **Agregar referencia**. En la página **.NET**, seleccione **Microsoft.Office.Interop.Word** en la lista **Nombre de componente**. Seleccione **Aceptar**.

Agregar las directivas using necesarias

En el **Explorador de soluciones**, haga clic con el botón derecho en el archivo *Program.cs* y luego seleccione **Ver código**. Agregue las directivas `using` siguientes a la parte superior del archivo de código:

```
C#  
  
using Word = Microsoft.Office.Interop.Word;
```

Mostrar texto en un documento de Word

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 parámetrosopcionales. En este ejemplo se usan los valores predeterminados. Por lo tanto, no se necesitan argumentos en la instrucción de llamada.

```
C#
```

```
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();
}
```

Agregue el código siguiente al final del método para definir dónde se muestra texto en el documento y qué texto se muestra:

C#

```
// 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. . .");
```

Ejecución de la aplicación

Agregue la instrucción siguiente a Principal:

C#

```
DisplayInWord();
```

Presione **CTRL** + **F5** para ejecutar el proyecto. Aparecerá un documento de Word con el texto especificado.

Cambiar el texto a una tabla

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. El argumento especifica que las comas de la cadena de texto de `range` separan las celdas de la tabla.

C#

```
// Convert to a simple table. The table will have a single row with  
// three columns.  
range.ConvertToTable(Separator: ",");
```

Presione `CTRL + F5` para ejecutar el proyecto.

Experimentar con otros parámetros

Cambie la tabla de modo que tenga una columna y tres filas, reemplace la última línea de `DisplayInWord` por la instrucción siguiente y, a continuación, escriba `CTRL + F5`.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

Para especificar un formato predefinido para la tabla, reemplace la última línea de `DisplayInWord` por la instrucción siguiente y después escriba `CTRL + F5`. El formato puede ser cualquiera de las constantes `WdTableFormat`.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,  
Format: Word.WdTableFormat.wdTableFormatElegant);
```

Ejemplo

En el código siguiente se incluye el ejemplo completo:

C#

```
using System;
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);  
}  
}  
}
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Uso del tipo dynamic

Artículo • 14/06/2023

El tipo `dynamic` es 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`. El compilador da por supuesto que un elemento `dynamic` admite cualquier operación. Por consiguiente, no tiene que determinar si el objeto obtiene su valor de una API COM, de un lenguaje dinámico como IronPython, del Document Object Model (DOM) de HTML, de la reflexión o de otro lugar en el programa. Sin embargo, si el código no es válido, se muestran errores en tiempo de ejecución.

Por ejemplo, si el método de instancia `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. Sin embargo, el error no pasa inadvertido indefinidamente. Aparece en tiempo de ejecución y genera una excepción en tiempo de ejecución.

C#

```
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();
}
```

C#

```
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 la expresión o el objeto `dynamic`. El entorno de ejecución examina la información almacenada y cualquier instrucción que no sea válida provoca 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`.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Entre las operaciones en las que el resultado no es de tipo `dynamic`, se 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`:

C#

```
var testInstance = new ExampleClass(d);
```

Conversiones

Las conversiones entre objetos dinámicos y de otro tipo son fáciles. Las conversiones permiten al desarrollador cambiar entre el comportamiento dinámico y no dinámico.

Puede convertir cualquiera en `dynamic` implícitamente, como se muestra en los ejemplos siguientes.

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Por el contrario, puede aplicar dinámicamente cualquier conversión implícita a cualquier expresión de tipo `dynamic`.

C#

```
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 toma 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.

C#

```
// 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) 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

Muchos métodos COM permiten variaciones en los tipos de argumento y el tipo de valor devuelto mediante la designación de los tipos como `object`. La interoperabilidad COM requiere la conversión explícita de los valores para coordinarse con variables fuertemente tipadas en C#. Si se compila con la opción [EmbedInteropTypes](#) ([opciones del compilador de C#](#)) la introducción del tipo `dynamic` le permite tratar las repeticiones de `object` en las signaturas de COM como si fueran de tipo `dynamic` y así evitar gran parte de la conversión. Para obtener más información sobre el uso del tipo `dynamic` con objetos COM, consulte el artículo [Acceso a los objetos de interoperabilidad de Office mediante características de C#](#).

Artículos relacionados

Title	Descripción
dynamic	Describe el uso de la palabra clave <code>dynamic</code> .
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 <code>IronPython</code> .

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Tutorial: Crear y usar objetos dinámicos en C#

Artículo • 09/05/2023

Los objetos dinámicos exponen miembros como propiedades y métodos en tiempo de ejecución, en lugar de en tiempo de compilación. Los objetos dinámicos le permiten 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 interpretado en tiempo de ejecución.

Para hacer referencia a un objeto dinámico, use un enlace en tiempo de ejecución. Especifique el tipo de un objeto enlazado en tiempo de ejecución como `dynamic`. Para obtener más información, consulte [dynamic](#).

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.

Este artículo contiene dos tutoriales independientes:

- 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

- Visual Studio 2022, versión 17.3 o posterior con la carga de trabajo **Desarrollo de escritorio de .NET** instalada. El SDK de .NET 7 se incluye al seleccionar esta carga de trabajo.

ⓘ Nota

Es posible que tu 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](#).

- En el segundo tutorial, instale [IronPython](#) para .NET. Vaya a su [página de descarga](#) para obtener la versión más reciente.

Creación de un objeto dinámico personalizado

En el primer tutorial se define un objeto dinámico personalizado que busca en el contenido de un archivo de texto. Una propiedad dinámica especifica el texto que se va a buscar. 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.

Crear una clase dinámica personalizada

Inicie Visual Studio. Seleccione **Crear un proyecto**. En el cuadro de diálogo **Crear un proyecto**, seleccione C#, después **Aplicación de consola** y luego **Siguiente**. En el cuadro de diálogo **Configurar el nuevo proyecto**, escriba `DynamicSample` como **Nombre del proyecto** y, después, seleccione **Siguiente**. En el cuadro de diálogo **Información**

adicional, seleccione .NET 7.0 (actual) para Plataforma de destino y después Crear. En el Explorador de soluciones, haga clic con el botón derecho en el proyecto DynamicSample y seleccione Agregar>Clase. En el cuadro Nombre, escriba `ReadOnlyFile` y, después, seleccione Agregar. 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`.

C#

```
using System.IO;
using System.Dynamic;
```

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.

C#

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

Actualice la instrucción de clase para heredar la clase `DynamicObject`, como se muestra en el ejemplo de código siguiente.

C#

```
class ReadOnlyFile : DynamicObject
```

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`.

C#

```
// 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;
}

```

1. Agregue el método `GetPropertyValues` siguiente a la clase `ReadOnlyFile`. El método `GetPropertyValues` 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 `GetPropertyValues` para recuperar los resultados correspondientes.

C#

```

public List<string> GetPropertyValues(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;
}

```

Después del método `GetPropertyValues`, 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`.

C#

```

// 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;
}

```

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 `GetPropertyValues` para recuperar los resultados.

C#

```
// 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 = GetPropertyValues(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

Guarde y cierre el archivo.

Creación de un archivo de texto de ejemplo

En el **Explorador de soluciones**, haga clic con el botón derecho en el proyecto `DynamicSample` y seleccione **Agregar>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 después seleccione **Agregar**. Copie el texto siguiente en el archivo *TextFile1.txt*.

```
text

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
```

Guarde y cierre el archivo.

Creación de una aplicación de ejemplo que usa el objeto dinámico personalizado

En el **Explorador de soluciones**, haga doble clic en el archivo *Program.cs*. 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".

```
C#

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);
}
```

Guarde el archivo y presione `Ctrl+F5` para compilar y ejecutar la aplicación.

Llamada a una biblioteca de lenguaje dinámico

En el tutorial siguiente se crea un proyecto que accede a una biblioteca escrita en el lenguaje dinámico IronPython.

Para crear una clase dinámica personalizada

Abra Visual Studio, seleccione **Archivo > Nuevo > Proyecto**. En el cuadro de diálogo **Crear un proyecto**, seleccione **C#**, después **Aplicación de consola** y luego **Siguiente**. En el cuadro de diálogo **Configurar el nuevo proyecto**, escriba `DynamicIronPythonSample` como **Nombre del proyecto** y, después, seleccione **Siguiente**. En el cuadro de diálogo **Información adicional**, seleccione **.NET 7.0 (actual)** para **Plataforma de destino** y después **Crear**. Instale el paquete NuGet [IronPython](#). Edite el archivo `Program.cs`. 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, y el espacio de nombres `System.Linq`.

```
C#
```

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

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`.

```
C#
```

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
    @"\IronPython 2.7\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.");
```

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.

C#

```
// 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("-----");  
}
```

Guarde el archivo y presione **Ctrl+F5** para compilar y ejecutar la aplicación.

Consulte también

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [Uso de tipo dinámico](#)
- [dynamic](#)
- [Implementar interfaces dinámicas \(PDF descargable de Microsoft TechNet\)](#) ↗

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Reduce memory allocations using new C# features

Article • 10/17/2023

ⓘ Important

The techniques described in this section improve performance when applied to *hot paths* in your code. *Hot paths* are those sections of your codebase that are executed often and repeatedly in normal operations. Applying these techniques to code that isn't often executed will have minimal impact. Before making any changes to improve performance, it's critical to measure a baseline. Then, analyze that baseline to determine where memory bottlenecks occur. You can learn about many cross platform tools to measure your application's performance in the section on [Diagnostics and instrumentation](#). You can practice a profiling session in the tutorial to [Measure memory usage](#) in the Visual Studio documentation.

Once you've measured memory usage and have determined that you can reduce allocations, use the techniques in this section to reduce allocations. After each successive change, measure memory usage again. Make sure each change has a positive impact on the memory usage in your application.

Performance work in .NET often means removing allocations from your code. Every block of memory you allocate must eventually be freed. Fewer allocations reduce time spent in garbage collection. It allows for more predictable execution time by removing garbage collections from specific code paths.

A common tactic to reduce allocations is to change critical data structures from `class` types to `struct` types. This change impacts the semantics of using those types. Parameters and returns are now passed by value instead of by reference. The cost of copying a value is negligible if the types are small, three words or less (considering one word being of natural size of one integer). It's measurable and can have real performance impact for larger types. To combat the effect of copying, developers can pass these types by `ref` to get back the intended semantics.

The C# `ref` features give you the ability to express the desired semantics for `struct` types without negatively impacting their overall usability. Prior to these enhancements, developers needed to resort to `unsafe` constructs with pointers and raw memory to achieve the same performance impact. The compiler generates *verifiably safe code* for the new `ref` related features. *Verifiably safe code* means the compiler detects possible

buffer overruns or accessing unallocated or freed memory. The compiler detects and prevents some errors.

Pass and return by reference

Variables in C# store *values*. In `struct` types, the value is the contents of an instance of the type. In `class` types, the value is a reference to a block of memory that stores an instance of the type. Adding the `ref` modifier means that the variable stores the *reference* to the value. In `struct` types, the reference points to the storage containing the value. In `class` types, the reference points to the storage containing the reference to the block of memory.

In C#, parameters to methods are *passed by value*, and return values are *return by value*. The *value* of the argument is passed to the method. The *value* of the return argument is the return value.

The `ref`, `in`, `ref readonly`, or `out` modifier indicates that the argument is *passed by reference*. A *reference* to the storage location is passed to the method. Adding `ref` to the method signature means the return value is *returned by reference*. A *reference* to the storage location is the return value.

You can also use *ref assignment* to have a variable refer to another variable. A typical assignment copies the *value* of the right hand side to the variable on the left hand side of the assignment. A *ref assignment* copies the memory location of the variable on the right hand side to the variable on the left hand side. The `ref` now refers to the original variable:

C#

```
int anInteger = 42; // assignment.  
ref int location = ref anInteger; // ref assignment.  
ref int sameLocation = ref location; // ref assignment  
  
Console.WriteLine(location); // output: 42  
  
sameLocation = 19; // assignment  
  
Console.WriteLine(anInteger); // output: 19
```

When you *assign* a variable, you change its *value*. When you *ref assign* a variable, you change what it refers to.

You can work directly with the storage for values using `ref` variables, pass by reference, and `ref` assignment. Scope rules enforced by the compiler ensure safety when working directly with storage.

The `ref readonly` and `in` modifiers both indicate that the argument should be passed by reference and can't be reassigned in the method. The difference is that `ref readonly` indicates that the method uses the parameter as a variable. The method might capture the parameter, or it might return the parameter by `readonly` reference. In those cases, you should use the `ref readonly` modifier. Otherwise, the `in` modifier offers more flexibility. You don't need to add the `in` modifier to an argument for an `in` parameter, so you can update existing API signatures safely using the `in` modifier. The compiler issues a warning if you don't add either the `ref` or `in` modifier to an argument for a `ref readonly` parameter.

Ref safe context

C# includes rules for `ref` expressions to ensure that a `ref` expression can't be accessed where the storage it refers to is no longer valid. Consider the following example:

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe context is the body of
CantEscape
}
```

The compiler reports an error because you can't return a reference to a local variable from a method. The caller can't access the storage being referred to. The *ref safe context* defines the scope in which a `ref` expression is safe to access or modify. The following table lists the *ref safe contexts* for variable types. `ref` fields can't be declared in a `class` or a non-ref `struct`, so those rows aren't in the table:

[Expand table](#)

Declaration	<i>ref safe context</i>
non-ref local	block where local is declared
non-ref parameter	current method
<code>ref</code> , <code>ref readonly</code> , <code>in</code> parameter	calling method

Declaration	<i>ref safe context</i>
<code>out</code> parameter	current method
<code>class</code> field	calling method
non-ref <code>struct</code> field	current method
<code>ref</code> field of <code>ref struct</code>	calling method

A variable can be `ref` returned if its *ref safe context* is the calling method. If its *ref safe context* is the current method or a block, `ref` return is disallowed. The following snippet shows two examples. A member field can be accessed from the scope calling a method, so a class or struct field's *ref safe context* is the calling method. The *ref safe context* for a parameter with the `ref`, or `in` modifiers is the entire method. Both can be `ref` returned from a member method:

```
C#
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

ⓘ Note

When the `ref readonly` or `in` modifier is applied to a parameter, that parameter can be returned by `ref readonly`, not `ref`.

The compiler ensures that a reference can't escape its *ref safe context*. You can use `ref` parameters, `ref return`, and `ref` local variables safely because the compiler detects if you've accidentally written code where a `ref` expression could be accessed when its storage isn't valid.

Safe context and ref structs

`ref struct` types require more rules to ensure they can be used safely. A `ref struct` type can include `ref` fields. That requires the introduction of a *safe context*. For most types, the *safe context* is the calling method. In other words, a value that's not a `ref struct` can always be returned from a method.

Informally, the *safe context* for a `ref struct` is the scope where all of its `ref` fields can be accessed. In other words, it's the intersection of the *ref safe context* of all its `ref` fields. The following method returns a `ReadOnlySpan<char>` to a member field, so its *safe context* is the method:

C#

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

In contrast, the following code emits an error because the `ref field` member of the `Span<int>` refers to the stack allocated array of integers. It can't escape the method:

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Unify memory types

The introduction of `System.Span<T>` and `System.Memory<T>` provide a unified model for working with memory. `System.ReadOnlySpan<T>` and `System.ReadOnlyMemory<T>` provide readonly versions for accessing memory. They all provide an abstraction over a block of memory storing an array of similar elements. The difference is that `Span<T>` and

`ReadOnlySpan<T>` are `ref struct` types whereas `Memory<T>` and `ReadOnlyMemory<T>` are `struct` types. Spans contain a `ref field`. Therefore instances of a span can't leave its *safe context*. The *safe context* of a `ref struct` is the *ref safe context* of its `ref field`. The implementation of `Memory<T>` and `ReadOnlyMemory<T>` remove this restriction. You use these types to directly access memory buffers.

Improve performance with ref safety

Using these features to improve performance involves these tasks:

- *Avoid allocations*: When you change a type from a `class` to a `struct`, you change how it's stored. Local variables are stored on the stack. Members are stored inline when the container object is allocated. This change means fewer allocations and that decreases the work the garbage collector does. It might also decrease memory pressure so the garbage collector runs less often.
- *Preserve reference semantics*: Changing a type from a `class` to a `struct` changes the semantics of passing a variable to a method. Code that modified the state of its parameters needs modification. Now that the parameter is a `struct`, the method is modifying a copy of the original object. You can restore the original semantics by passing that parameter as a `ref` parameter. After that change, the method modifies the original `struct` again.
- *Avoid copying data*: Copying larger `struct` types can impact performance in some code paths. You can also add the `ref` modifier to pass larger data structures to methods by reference instead of by value.
- *Restrict modifications*: When a `struct` type is passed by reference, the called method could modify the state of the struct. You can replace the `ref` modifier with the `ref readonly` or `in` modifiers to indicate that the argument can't be modified. Prefer `ref readonly` when the method captures the parameter or returns it by readonly reference. You can also create `readonly struct` types or `struct` types with `readonly` members to provide more control over what members of a `struct` can be modified.
- *Directly manipulate memory*: Some algorithms are most efficient when treating data structures as a block of memory containing a sequence of elements. The `Span` and `Memory` types provide safe access to blocks of memory.

None of these techniques require `unsafe` code. Used wisely, you can get performance characteristics from safe code that was previously only possible by using unsafe techniques. You can try the techniques yourself in the tutorial on [reducing memory allocations](#).

Tutorial: Reducción de asignaciones de memoria con seguridad `ref`

Artículo • 13/10/2023

A menudo, el ajuste del rendimiento de una aplicación .NET implica dos técnicas. Primero, la reducción del número y el tamaño de las asignaciones del montón. Segundo, la reducción de la frecuencia con la que se copian los datos. Visual Studio proporciona excelentes [herramientas](#) que ayudan a analizar cómo usa la aplicación la memoria. Una vez que haya determinado dónde realiza la aplicación asignaciones innecesarias, realice cambios para minimizar esas asignaciones. Los tipos `class` se convierten en tipos `struct`. Use [características](#) de seguridad `ref` para conservar la semántica y minimizar la copia adicional.

Use [Visual Studio 17.5](#) para obtener la mejor experiencia con este tutorial. La herramienta de asignación de objetos .NET que se usa para analizar el uso de memoria forma parte de Visual Studio. Puede usar [Visual Studio Code](#) y la línea de comandos para ejecutar la aplicación y realizar todos los cambios. Pero no podrá ver los resultados del análisis de los cambios.

La aplicación que usará es una simulación de una aplicación de IoT que supervisa varios sensores para determinar si un intruso ha introducido una galería secreta con objetos valiosos. Los sensores de IoT envían constantemente datos que miden la combinación de oxígeno (O₂) y dióxido de carbono (CO₂) en el aire. También informan de la temperatura y la humedad relativa. Cada uno de estos valores fluctúa ligeramente todo el tiempo. Pero cuando una persona entra en la habitación, cambia un poco más y siempre en la misma dirección: el oxígeno disminuye, el dióxido de carbono aumenta, y tanto la temperatura como la humedad relativa aumentan también. Cuando los sensores se combinan para mostrar aumentos, se desencadena la alarma de intruso.

En este tutorial, ejecutará la aplicación, tomará medidas de las asignaciones de memoria y, después, mejorará el rendimiento reduciendo el número de asignaciones. El código fuente está disponible en el [explorador de ejemplos](#).

Exploración de la aplicación de inicio

Descargue la aplicación y ejecute el ejemplo de inicio. La aplicación de inicio funciona correctamente, pero como asigna muchos objetos pequeños con cada ciclo de medida, su rendimiento se degrada lentamente según se ejecuta con el tiempo.

Consola

Press <return> to start simulation

Debounced measurements:

Temp: 67.332
Humidity: 41.077%
Oxygen: 21.097%
CO2 (ppm): 404.906

Average measurements:

Temp: 67.332
Humidity: 41.077%
Oxygen: 21.097%
CO2 (ppm): 404.906

Debounced measurements:

Temp: 67.349
Humidity: 46.605%
Oxygen: 20.998%
CO2 (ppm): 408.707

Average measurements:

Temp: 67.349
Humidity: 46.605%
Oxygen: 20.998%
CO2 (ppm): 408.707

Se han quitado muchas filas.

Consola

Debounced measurements:

Temp: 67.597
Humidity: 46.543%
Oxygen: 19.021%
CO2 (ppm): 429.149

Average measurements:

Temp: 67.568
Humidity: 45.684%
Oxygen: 19.631%
CO2 (ppm): 423.498

Current intruders: 3

Calculated intruder risk: High

Debounced measurements:

Temp: 67.602
Humidity: 46.835%
Oxygen: 19.003%
CO2 (ppm): 429.393

Average measurements:

Temp: 67.568
Humidity: 45.684%
Oxygen: 19.631%
CO2 (ppm): 423.498

```
Current intruders: 3  
Calculated intruder risk: High
```

Puede explorar el código para obtener información sobre cómo funciona la aplicación. El programa principal ejecuta la simulación. Después de presionar `<Enter>`, crea una sala y recopila algunos datos iniciales de línea de base:

C#

```
Console.WriteLine("Press <return> to start simulation");  
Console.ReadLine();  
var room = new Room("gallery");  
var r = new Random();  
  
int counter = 0;  
  
room.TakeMeasurements(  
    m =>  
    {  
        Console.WriteLine(room.Debounce);  
        Console.WriteLine(room.Average);  
        Console.WriteLine();  
        counter++;  
        return counter < 20000;  
});
```

Una vez que se han establecido los datos de línea de base, ejecuta la simulación en la sala, donde un generador de números aleatorios determina si un intruso ha entrado en la sala:

C#

```
counter = 0;  
room.TakeMeasurements(  
    m =>  
    {  
        Console.WriteLine(room.Debounce);  
        Console.WriteLine(room.Average);  
        room.Intruders += (room.Intruders, r.Next(5)) switch  
        {  
            ( > 0, 0 ) => -1,  
            ( < 3, 1 ) => 1,  
            _ => 0  
        };  
  
        Console.WriteLine($"Current intruders: {room.Intruders}");  
        Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");  
        Console.WriteLine();  
        counter++;  
    });
```

```
        return counter < 200000;  
});
```

Otros tipos contienen las medidas, una medida sin rebotes, que es el promedio de las últimas 50 medidas, y el promedio de todas las medidas tomadas.

Ahora, ejecute la aplicación mediante la [herramienta de asignación de objetos de .NET](#). Asegúrese de usar la compilación `Release`, no la `Debug`. En el menú *Depurar*, abra el *Generador de perfiles de rendimiento*. Active la opción *Seguimiento de asignación de objetos de .NET*, pero nada más. Ejecute la aplicación hasta su finalización. El generador de perfiles mide las asignaciones de objetos e informa sobre las asignaciones y los ciclos de recolección de elementos no utilizados. Debería ver un gráfico similar al de la imagen siguiente:



En el gráfico anterior se muestra que trabajar para minimizar las asignaciones proporcionará ventajas de rendimiento. Verá un patrón de sierra en el gráfico de objetos activos. Esto le indica que se crean numerosos objetos que se convierten rápidamente en elementos no utilizados. Más adelante se recopilan, como se muestra en el gráfico delta del objeto. Las barras rojas descendentes indican un ciclo de recolección de elementos no utilizados.

Ahora, examine la pestaña *Asignaciones* debajo de los gráficos. En esta tabla se muestran los tipos que más se asignan:

Type	Allocations
Small Object Heap	
System.String	842,451
IntruderAlert.SensorMeasurement	220,000
IntruderAlert.IntruderRisk	200,000
System.Char[]	80
System.Diagnostics.Tracing.EventSource.EventMetadata[]	7
System.Byte[]	339

El tipo `System.String` cuenta con la mayoría de las asignaciones. La tarea más importante debe ser minimizar la frecuencia de las asignaciones de cadenas. Esta aplicación imprime numerosas salidas con formato en la consola de forma constante. Para esta simulación, queremos mantener los mensajes, por lo que nos centraremos en las dos filas siguientes: los tipos `SensorMeasurement` y `IntruderRisk`.

Haga doble clic en la línea `SensorMeasurement`. Puede ver que todas las asignaciones tienen lugar en el método `static SensorMeasurement.TakeMeasurement`. Puede ver el método en el fragmento de código siguiente:

```
C#  
  
public static SensorMeasurement TakeMeasurement(string room, int intruders)  
{  
    return new SensorMeasurement  
    {  
        CO2 = (CO2Concentration + intruders * 10) + (20 *  
generator.NextDouble() - 10.0),  
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *  
generator.NextDouble() - 0.0025),  
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *  
generator.NextDouble() - 0.25),  
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *  
generator.NextDouble() - 0.10),  
        Room = room,  
        TimeRecorded = DateTime.Now  
    };  
}
```

Cada medida asigna un objeto `SensorMeasurement` nuevo, que es tipo `class`. Cada `SensorMeasurement` creado provoca una asignación de montón.

Cambio de clases a estructuras

El código siguiente muestra la declaración inicial de `SensorMeasurement`:

```
C#  
  
public class SensorMeasurement  
{  
    private static readonly Random generator = new Random();  
  
    public static SensorMeasurement TakeMeasurement(string room, int  
intruders)  
    {  
        return new SensorMeasurement  
        {
```

```

        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}

private const double CO2Concentration = 409.8; // increases with people.
private const double O2Concentration = 0.2100; // decreases
private const double TemperatureSetting = 67.5; // increases
private const double HumiditySetting = 0.4500; // increases

public required double CO2 { get; init; }
public required double O2 { get; init; }
public required double Temperature { get; init; }
public required double Humidity { get; init; }
public required string Room { get; init; }
public required DateTime TimeRecorded { get; init; }

public override string ToString() => $"""
    Room: {Room} at {TimeRecorded}:
        Temp:      {Temperature:F3}
        Humidity: {Humidity:P3}
        Oxygen:   {O2:P3}
        CO2 (ppm): {CO2:F3}
    """;
}

```

El tipo se creó originalmente como `class` porque contiene numerosas medidas `double`. Es mayor de lo que desea copiar en rutas de acceso activas. Pero esa decisión implicaba un gran número de asignaciones. Cambie el tipo de `class` a `struct`.

Al cambiar de `class` a `struct`, se presentan algunos errores del compilador porque el código original usó comprobaciones de referencia `null` en algunos puntos. La primera está en la clase `DebounceMeasurement`, en el método `AddMeasurement`:

C#

```

public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
}

```

```

        double sumTemp = 0;
        double sumHumidity = 0;
        for (int i = 0; i < debounceSize; i++)
        {
            if (recentMeasurements[i] is not null)
            {
                sumCO2 += recentMeasurements[i].CO2;
                sumO2+= recentMeasurements[i].O2;
                sumTemp+= recentMeasurements[i].Temperature;
                sumHumidity += recentMeasurements[i].Humidity;
            }
        }
        O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
        totalMeasurements);
        CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
        totalMeasurements);
        Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
        debounceSize : totalMeasurements);
        Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
        debounceSize : totalMeasurements);
    }
}

```

El tipo `DebounceMeasurement` contiene una matriz de 50 medidas. Las lecturas para un sensor se informan como el promedio de las últimas 50 medidas. Esto reduce el ruido en las lecturas. Antes de tomar 50 lecturas completas, estos valores son `null`. El código comprueba si hay una referencia `null` para informar del promedio correcto en el startup. Después de cambiar el tipo `SensorMeasurement` a una estructura, debe usar una prueba diferente. El tipo `SensorMeasurement` incluye `string` para el identificador de sala, por lo que puede usar esa prueba en su lugar:

C#

```
if (recentMeasurements[i].Room is not null)
```

Los otros tres errores del compilador están todos en el método que toma repetidamente medidas en una sala:

C#

```

public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
{
    SensorMeasurement? measure = default;
    do {
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    }
}

```

```
    } while (MeasurementHandler(measure));  
}
```

En el método inicial, la variable local de `SensorMeasurement` es una *referencia que admite un valor NULL*:

C#

```
SensorMeasurement? measure = default;
```

Ahora que `SensorMeasurement` es `struct` en lugar de `class`, el que admite un valor NULL es un *tipo de valor que admite un valor NULL*. Puede cambiar la declaración a un tipo de valor para corregir los errores restantes del compilador:

C#

```
SensorMeasurement measure = default;
```

Ahora que se han solucionado los errores del compilador, debe examinar el código para asegurarse de que la semántica no ha cambiado. Dado que los tipos `struct` se pasan por valor, las modificaciones realizadas en los parámetros del método no son visibles después de que el método devuelva.

ⓘ Importante

Cambiar un tipo de `class` a `struct` puede cambiar la semántica del programa. Cuando un tipo `class` se pasa a un método, las mutaciones realizadas en el método se realizan en el argumento. Cuando un tipo `struct` se pasa a un método y las mutaciones realizadas en el método se realizan en *una copia* del argumento. Esto significa que cualquier método que modifique sus argumentos por diseño debe actualizarse para usar el modificador `ref` en cualquier tipo de argumento que haya cambiado de `class` a `struct`.

El tipo `SensorMeasurement` no incluye ningún método que cambie el estado, por lo que no es un problema en este ejemplo. Puede demostrarlo agregando el modificador `readonly` a la estructura `SensorMeasurement`:

C#

```
public readonly struct SensorMeasurement
```

El compilador aplica la naturaleza `readonly` de struct `SensorMeasurement`. Si la inspección del código pasara por alto algún método que modifica el estado, el compilador le avisaría. La aplicación sigue compilando sin errores, por lo que este tipo es `readonly`. Agregar el modificador `readonly` al cambiar un tipo de `class` a `struct` puede ayudarle a encontrar miembros que modifiquen el estado de `struct`.

Evitación de la realización de copias

Ha quitado un gran número de asignaciones innecesarias de la aplicación. El tipo `SensorMeasurement` no aparece en ningún lugar de la tabla.

Ahora, está realizando un trabajo adicional al copiar la estructura `SensorMeasurement` cada vez que se usa como parámetro o valor devuelto. La estructura `SensorMeasurement` contiene cuatro dobles, una unidad de `DateTime` y otra de `string`. Esa estructura es considerablemente mayor que una referencia. Vamos a agregar los modificadores `ref` o `in` a lugares donde se usa el tipo `SensorMeasurement`.

El siguiente paso consiste en buscar métodos que devuelven una medida o toman una medida como argumento y usan referencias siempre que sea posible. Comience en la estructura `SensorMeasurement`. El método estático `TakeMeasurement` crea y devuelve una medida `SensorMeasurement` nueva:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Lo dejaremos tal como está, devolviendo por valor. Si intentó devolver por `ref`, obtendría un error del compilador. No se puede devolver un elemento `ref` a una estructura nueva creada localmente en el método. El diseño de la estructura inmutable

implica que solo se pueden establecer los valores de la medida en la construcción. Este método debe crear una estructura de medida nueva.

Volvamos a echar un vistazo a `DebounceMeasurement.AddMeasurement`. Debe agregar el modificador `in` al parámetro `measurement`:

C#

```
public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize : totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize : totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? debounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ? debounceSize : totalMeasurements);
}
```

Eso guarda una operación de copia. El parámetro `in` es una referencia a la copia ya creada por el autor de la llamada. También puede guardar una copia con el método `TakeMeasurement` en el tipo `Room`. Este método muestra cómo el compilador proporciona seguridad al pasar argumentos por `ref`. El método inicial `TakeMeasurement` del tipo `Room` toma un argumento de `Func<SensorMeasurement, bool>`. Si intenta agregar el modificador `in` o `ref` a esa declaración, el compilador notifica un error. No se puede pasar un argumento `ref` a una expresión lambda. El compilador no puede garantizar que la expresión llamada no copie la referencia. Si la expresión lambda *captura* la referencia, esta podría tener una duración mayor que el valor al que hace referencia. Acceder a él fuera de su contexto seguro `ref` provocaría daños en la memoria. Las reglas

de seguridad `ref` no lo permiten. Puede obtener más información en la introducción a las características de seguridad `ref`.

Conservación de la semántica

Los conjuntos finales de cambios no tendrán un impacto importante en el rendimiento de esta aplicación porque los tipos no se crean en rutas de acceso activas. Estos cambios muestran algunas de las otras técnicas que usaría en el ajuste del rendimiento. Echemos un vistazo a la clase inicial `Room`:

```
C#  
  
public class Room  
{  
    public AverageMeasurement Average { get; } = new ();  
    public DebounceMeasurement Debounce { get; } = new ();  
    public string Name { get; }  
  
    public IntruderRisk RiskStatus  
    {  
        get  
        {  
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 / 4;  
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 / 4.0;  
            var TempVariance = (Debounce.Temperature - Average.Temperature)  
> 0.05 / 4.0;  
            var HumidityVariance = (Debounce.Humidity - Average.Humidity) >  
0.20 / 4;  
            IntruderRisk risk = IntruderRisk.None;  
            if (CO2Variance) { risk++; }  
            if (O2Variance) { risk++; }  
            if (TempVariance) { risk++; }  
            if (HumidityVariance) { risk++; }  
            return risk;  
        }  
    }  
  
    public int Intruders { get; set; }  
  
    public Room(string name)  
    {  
        Name = name;  
    }  
  
    public void TakeMeasurements(Func<SensorMeasurement, bool>  
MeasurementHandler)  
    {  
        SensorMeasurement? measure = default;  
        do {  
            measure = SensorMeasurement.TakeMeasurement(Name, Intruders);  
        } while (measure != null && MeasurementHandler(measure));  
    }  
}
```

```
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    } while (MeasurementHandler(measure));
}
}
```

Este tipo contiene varias propiedades. Algunos son tipos `class`. La creación de un objeto `Room` implica varias asignaciones. Uno para el propio objeto `Room` y otro para cada uno de los miembros de tipo `class` que contiene. Puede convertir dos de estas propiedades de tipos `class` a `struct`: los tipos `DebounceMeasurement` y `AverageMeasurement`. Vamos a trabajar con esa transformación con ambos tipos.

Cambie el tipo `DebounceMeasurement` de `class` a `struct`. Esto introduce un error del compilador `CS8983: A 'struct' with field initializers must include an explicitly declared constructor`. Para corregirlo, agregue un constructor sin parámetros vacío:

C#

```
public DebounceMeasurement() { }
```

Puede obtener más información sobre este requisito en el artículo de referencia del lenguaje sobre [estructuras](#).

La invalidación `Object.ToString()` no modifica ninguno de los valores de la estructura. Puede agregar el modificador `readonly` a esa declaración de método. El tipo `DebounceMeasurement` es *mutable*, por lo que deberá evitar que las modificaciones afecten a las copias que se descartan. El método `AddMeasurement` modifica el estado del objeto. Se llama desde la clase `Room`, en el método `TakeMeasurements`. Quiere que esos cambios se conserven después de llamar al método. Puede cambiar la propiedad `Room.Debounce` para devolver una *referencia* a una sola instancia del tipo `DebounceMeasurement`:

C#

```
private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref
debounce; } }
```

Hay algunos cambios en el ejemplo anterior. En primer lugar, la *propiedad* es una propiedad `readonly` que devuelve una referencia de solo lectura a la instancia de propiedad de esta sala. Ahora está respaldado por un campo declarado que se inicializa cuando se crea una instancia del objeto `Room`. Después de realizar estos cambios,

actualizará la implementación del método `AddMeasurement`. Usa el campo de respaldo privado, `debounce`, no la propiedad `readonly Debounce`. De este modo, los cambios tienen lugar en la única instancia creada durante la inicialización.

La misma técnica funciona con la propiedad `Average`. Primero, modifique el tipo `AverageMeasurement` de `class` a `struct` y agregue el modificador `readonly` en el método `ToString`:

C#

```
namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
    private double sumO2 = 0;
    private double sumTemperature = 0;
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
    public readonly double Temperature => sumTemperature /
totalMeasurements;
    public readonly double Humidity => sumHumidity / totalMeasurements;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
        Average measurements:
            Temp:      {Temperature:F3}
            Humidity: {Humidity:P3}
            Oxygen:   {O2:P3}
            CO2 (ppm): {CO2:F3}
        """;;
}
```

Después, modifique la clase `Room` siguiendo la misma técnica que usó para la propiedad `Debounce`. La propiedad `Average` devuelve un objeto `readonly ref` al campo privado para la medición promedio. El método `AddMeasurement` modifica los campos internos.

C#

```
private AverageMeasurement average = new();
public ref readonly AverageMeasurement Average { get { return ref average;
} }
```

Evasión de la conversión boxing

Hay un último cambio para mejorar el rendimiento. El programa principal está imprimiendo estadísticas para la sala, incluida la evaluación de riesgos:

C#

```
Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
```

La llamada a los cuadros `ToString` generados establece el valor de enumeración. Puede evitarlo escribiendo una invalidación en la clase `Room` que da formato a la cadena en función del valor de riesgo estimado:

C#

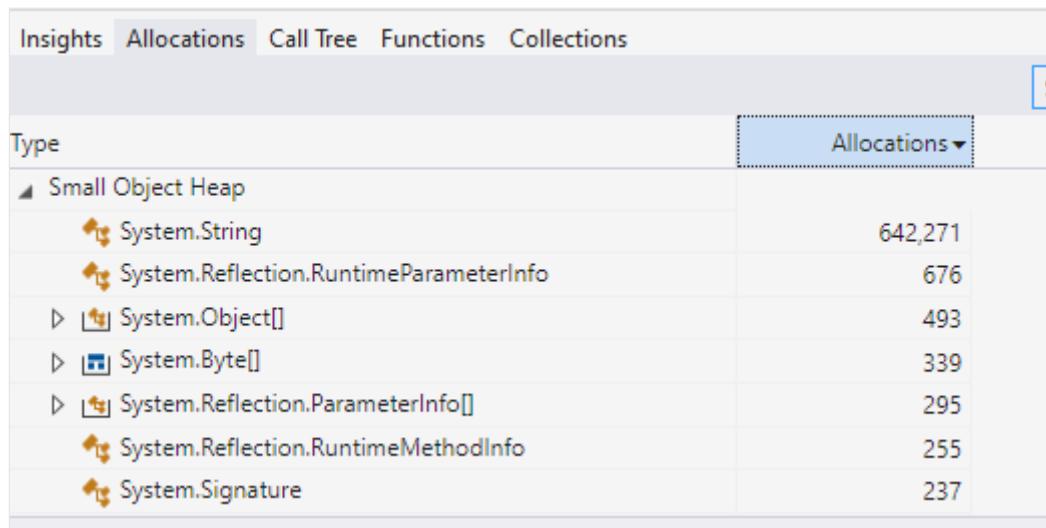
```
public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";
```

Después, modifique el código del programa principal para llamar a este método `ToString` nuevo:

C#

```
Console.WriteLine(room.ToString());
```

Ejecute la aplicación con el generador de perfiles y examine la tabla actualizada para las asignaciones.



Ha quitado numerosas asignaciones y ha proporcionado a la aplicación un aumento del rendimiento.

Uso de la seguridad ref en la aplicación

Estas técnicas son un ajuste de rendimiento de bajo nivel. Pueden aumentar el rendimiento de la aplicación cuando se aplican a las rutas de acceso activas y cuando se ha medido el impacto antes y después de los cambios. En la mayoría de los casos, el ciclo que seguirá es el siguiente:

- *Asignaciones de medida*: determine qué tipos se asignan más y cuándo puede reducir las asignaciones del montón.
- *Convertir la clase en estructura*: muchas veces, los tipos se pueden convertir de `class` a `struct`. La aplicación usa espacio de pila en lugar de realizar asignaciones de montón.
- *Conservar la semántica*: la conversión de `class` a `struct` puede afectar a la semántica de los parámetros y los valores devueltos. Cualquier método que modifique sus parámetros ahora debe marcarlos con el modificador `ref`. Esto garantiza que las modificaciones se realizan en el objeto correcto. Del mismo modo, si el autor de la llamada debe modificar una propiedad o un valor devuelto de método, dicho valor devuelto debe marcarse con el modificador `ref`.
- *Evitar copias*: cuando se pasa una estructura grande como parámetro, puede marcarlo con el modificador `in`. Puede pasar una referencia en menos bytes y asegurarse de que el método no modifica el valor original. También puede devolver valores con `readonly ref` para devolver una referencia que no se pueda modificar.

Con estas técnicas, puede mejorar el rendimiento en las rutas de acceso activas del código.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

SDK de .NET Compiler Platform

Artículo • 11/04/2024

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. Es la misión principal de las API de Roslyn: abrir las cajas negras y permitir que herramientas y usuarios finales compartan la gran cantidad de información que los compiladores tienen sobre el código. En lugar de actuar como traductores opacos de código fuente de entrada en código objeto de salida, con Roslyn, los compiladores se convierten en plataformas: 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 (estructura del código) y la semántica, y detectan las prácticas que se deben corregir. Las *correcciones de código* sugieren una o varias correcciones para tratar los errores de codificación que encuentran los analizadores o los diagnósticos del compilador. 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.

Además de los analizadores y las correcciones de código, el SDK de .NET Compiler Platform también le permite compilar *refactorizaciones de código*. También 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.

💡 Sugerencia

Antes de compilar su propio analizador, consulte los integrados. Para obtener más información, vea [Reglas de estilo del código](#).

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. Selección de **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. Selección de **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**.

Comentarios

¿Le ha resultado útil esta página? 👍 Sí 👎 No

[Proporcionar comentarios sobre el producto](#) 

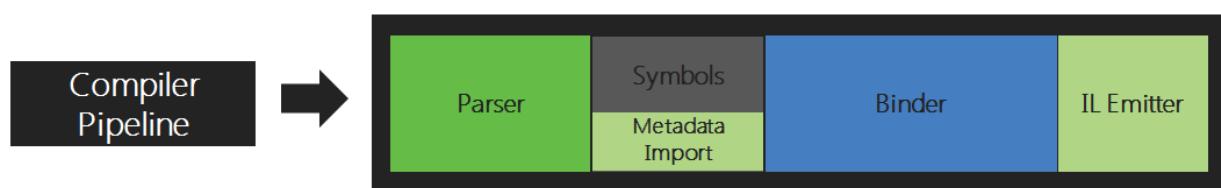
Descripción del modelo del SDK de .NET Compiler Platform

Artículo • 10/05/2023

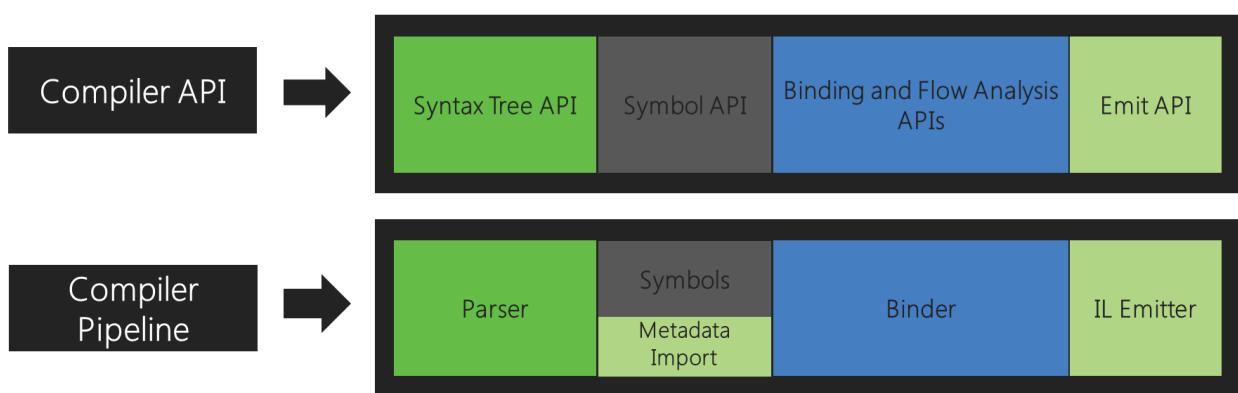
Los compiladores procesan el código que se escribe conforme a reglas estructuradas que a menudo se diferencian de la forma en que los seres humanos lo leen y lo entienden. Una comprensión básica del modelo usado por los compiladores resulta fundamental para entender las API que se usan al compilar herramientas basadas en Roslyn.

Áreas funcionales de canalización de compilador

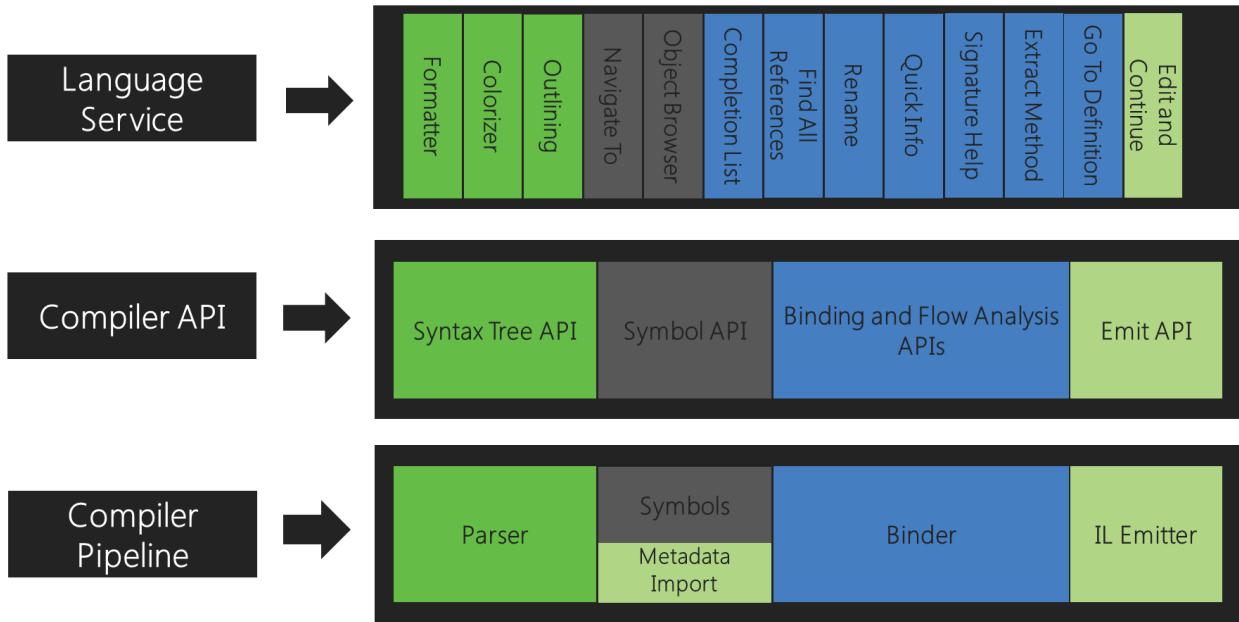
El SDK de .NET Compiler Platform expone el análisis de código de los compiladores de C# y Visual Basic al proporcionar una capa de API que refleja una canalización de compilador tradicional.



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 **Ira definición** usan el modelo semántico, y **Editar y continuar** usa todos ellos, incluida la API de emisión.

Capas de API

El SDK del compilador de .NET consta de varias capas de API: de compilador, de diagnóstico, de scripting y de área 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. Las 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, 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 activos 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

Artículo • 09/05/2023

El *árbol de sintaxis* es una estructura de datos fundamental e inmutable 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:

- Contienen toda la información de origen con fidelidad completa. 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.
- Pueden generar el texto exacto mediante 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, de manera implícita, ha creado el texto equivalente, mientras que, al crear uno a partir de los cambios en uno existente, ha editado realmente el texto.
- Son inmutables y seguros para subprocessos. 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:

```
C#  
  
    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.

💡 Sugerencia

Se recomienda comprobar los tipos con los métodos de extensión [IsKind](#) (para C#) o [IsKind](#) (para VB).

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.IsMatched](#) 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

Artículo • 10/05/2023

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 que devuelve el 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

Artículo • 10/05/2023

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 y documentos

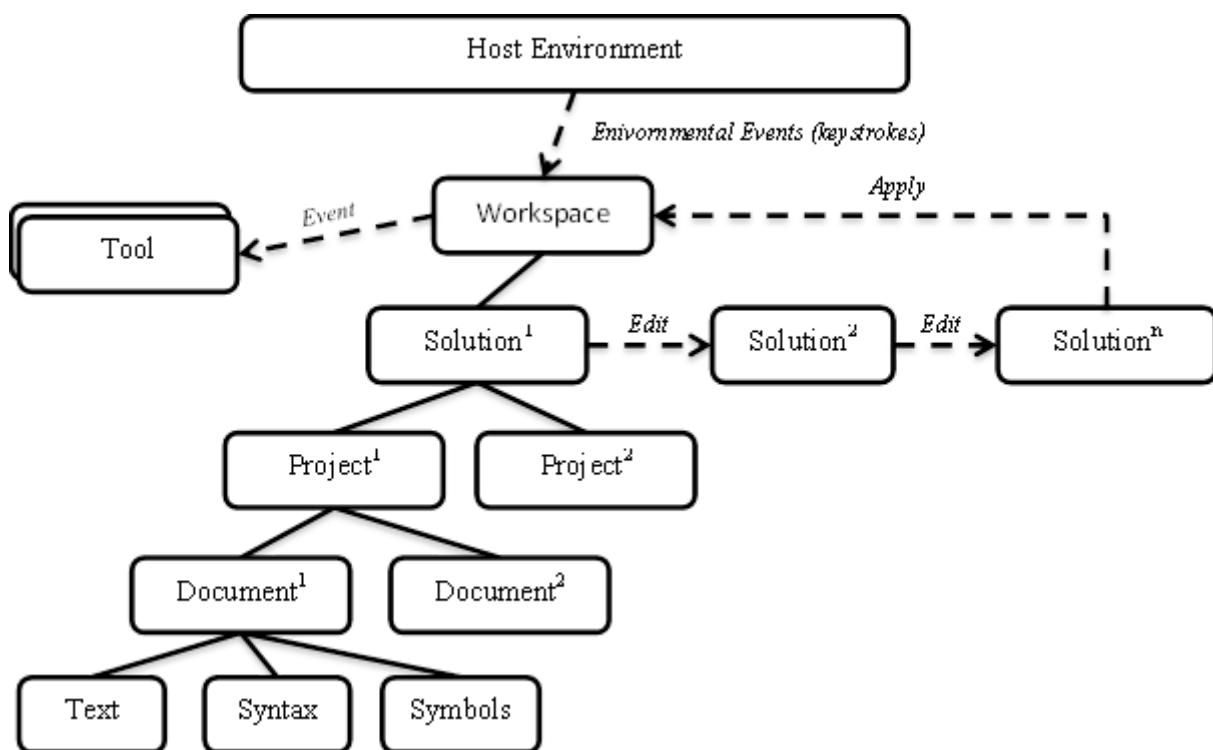
Aunque un área de trabajo puede cambiar cada vez que se pulsa una tecla, puede trabajar con el modelo de la solución de forma aislada.

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

Artículo • 10/05/2023

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. Selección de **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 interesaría 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. Selección de **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:

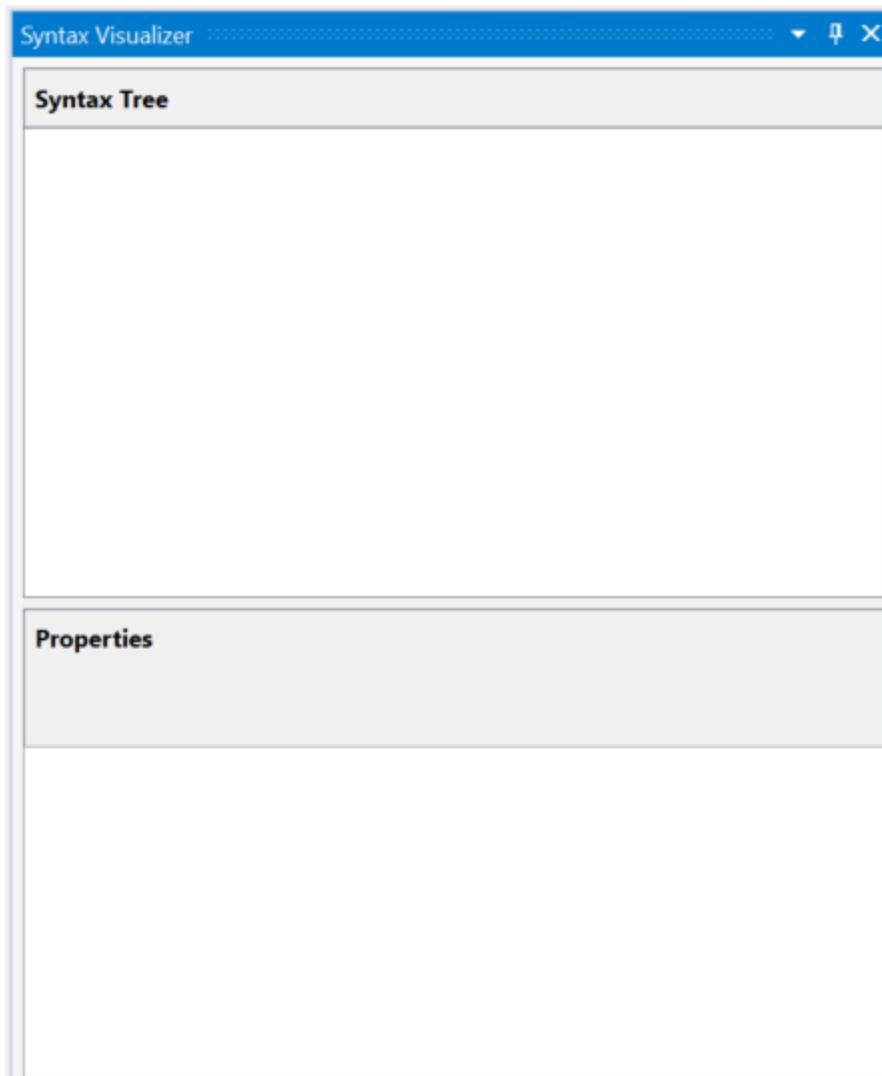
1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

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**.

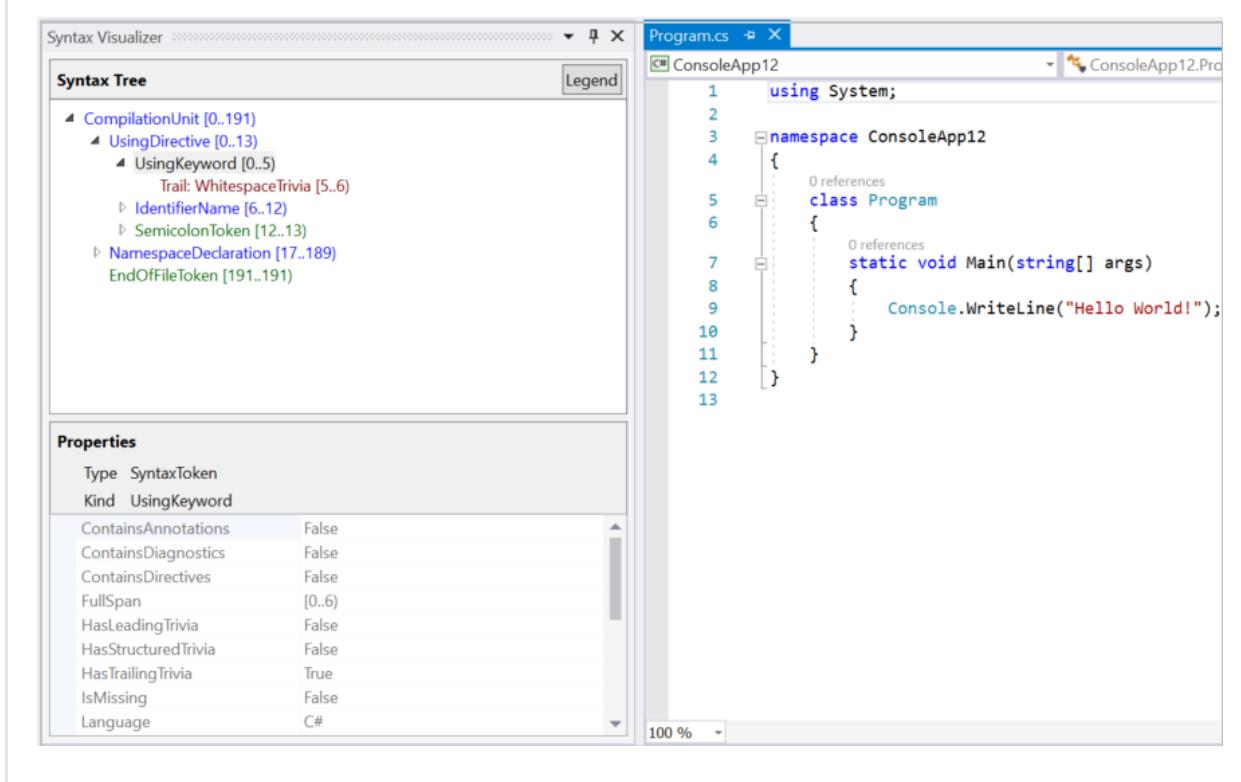
Este comando abre el Visualizador de sintaxis como una ventana de herramientas flotante. Si no tiene abierta ninguna ventana de editor de código, la presentación está en blanco, tal como se muestra en esta imagen.



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#



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.

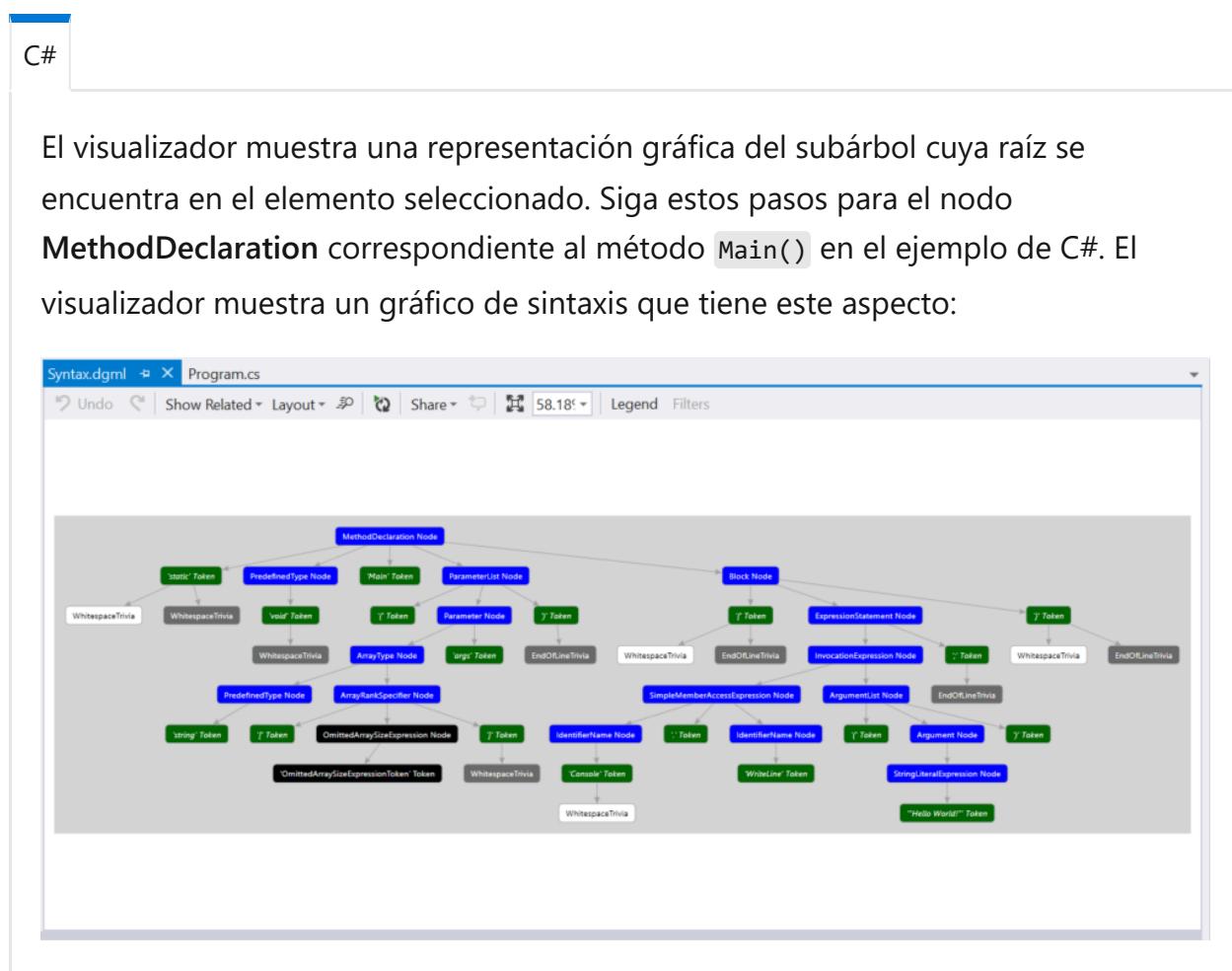
El visualizador resalta el elemento en el árbol cuyo intervalo coincide mejor con el intervalo del texto seleccionado en el editor.

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

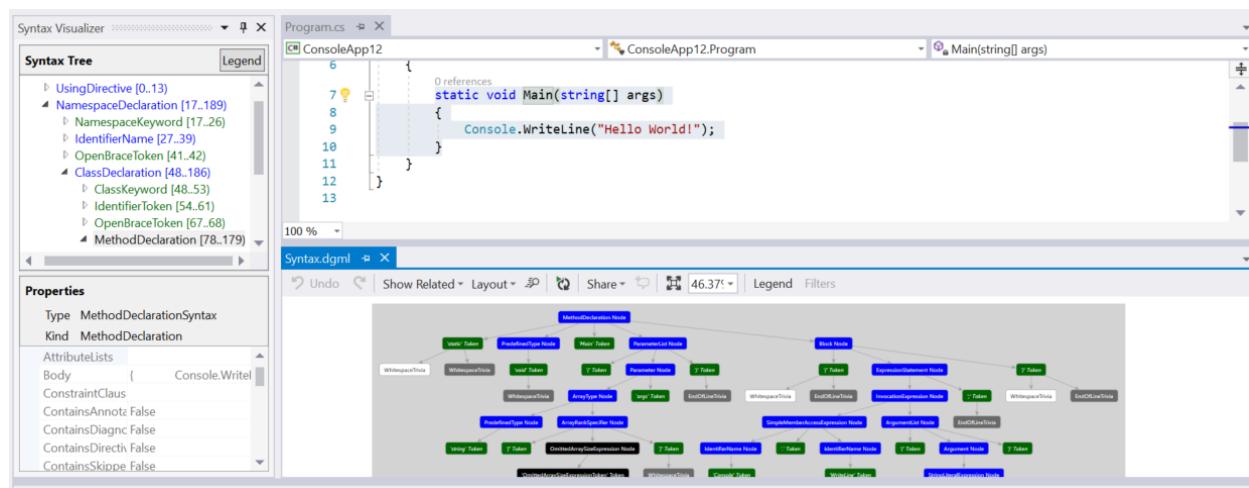
Haga clic con el botón derecho en cualquier elemento del árbol y haga clic en **View Directed Syntax Graph** (Ver gráfico de sintaxis dirigido).



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**.

The screenshot shows the Syntax Visualizer window with the following details:

- Syntax Tree:** Shows the hierarchical structure of the code. The `AddExpression` node is selected.
- Properties:**
 - Type: `SynthesizedIntrinsicOperatorSymbol`
 - Kind: `Method`
 - Name: `op_Addition`
 - OriginalDefinition: `int.operator +(int, int)`
 - OverriddenMethod: (Collection)
 - Parameters: (Collection)
 - PartialDefinitionPart: (Collection)
 - PartialImplementation: (Collection)
 - ReceivingType: `int`

Red arrows point from the text "Intente View TypeSymbol (if any) [Ver TypeSymbol (si existe)] para el mismo nodo AddExpression." to the "Type" and "Kind" fields in the Properties grid.

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`.

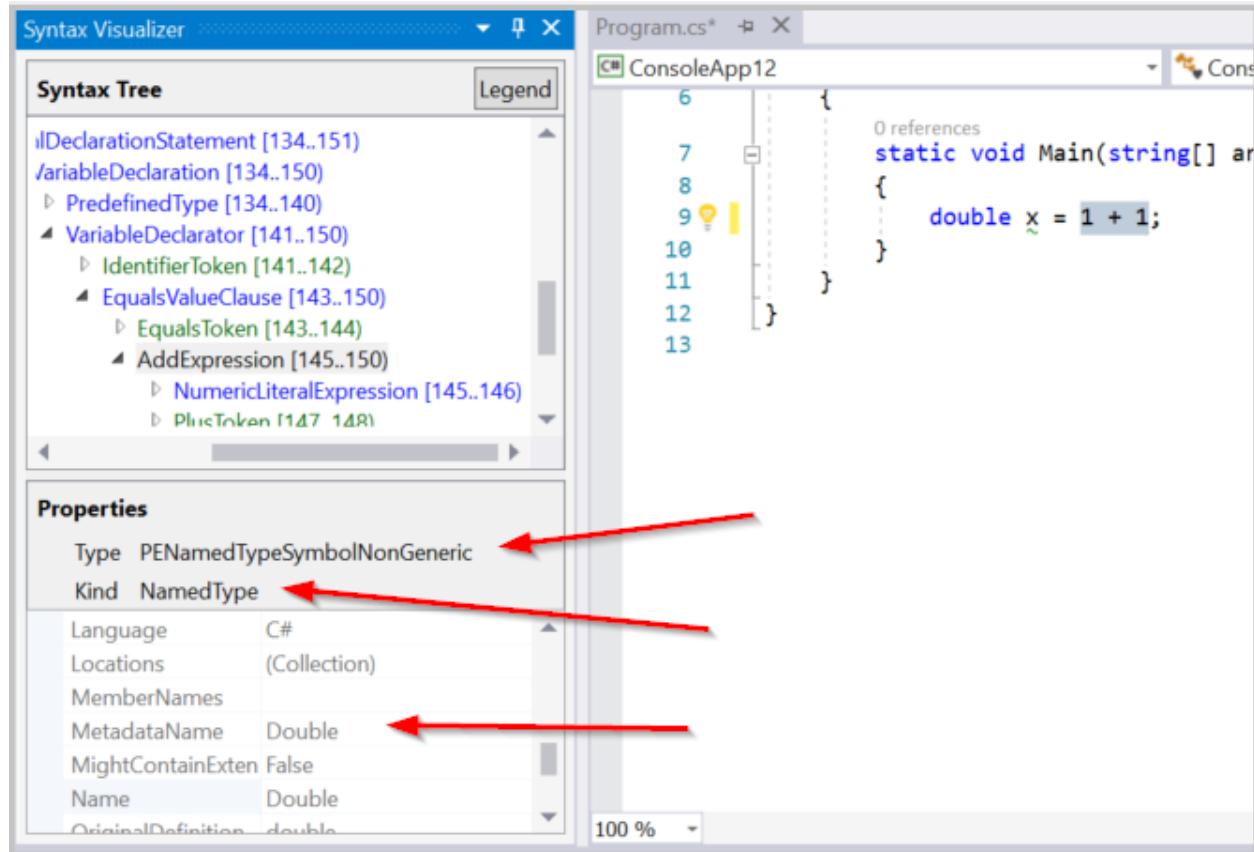
The screenshot shows the Syntax Visualizer window with the following details:

- Syntax Tree:** Shows the hierarchical structure of the code. The `AddExpression` node is selected.
- Properties:**
 - Type: `PENamedTypeSymbolNonGeneric`
 - Kind: `NamedType`
 - Name: `Int32`
 - OriginalDefinition: `int`
 - SpecialType: `System_Int32`
 - StaticConstructors: (Collection)
 - TupleElementName: (Collection)
 - TupleElements: (Collection)
 - TupleElementTypes: (Collection)

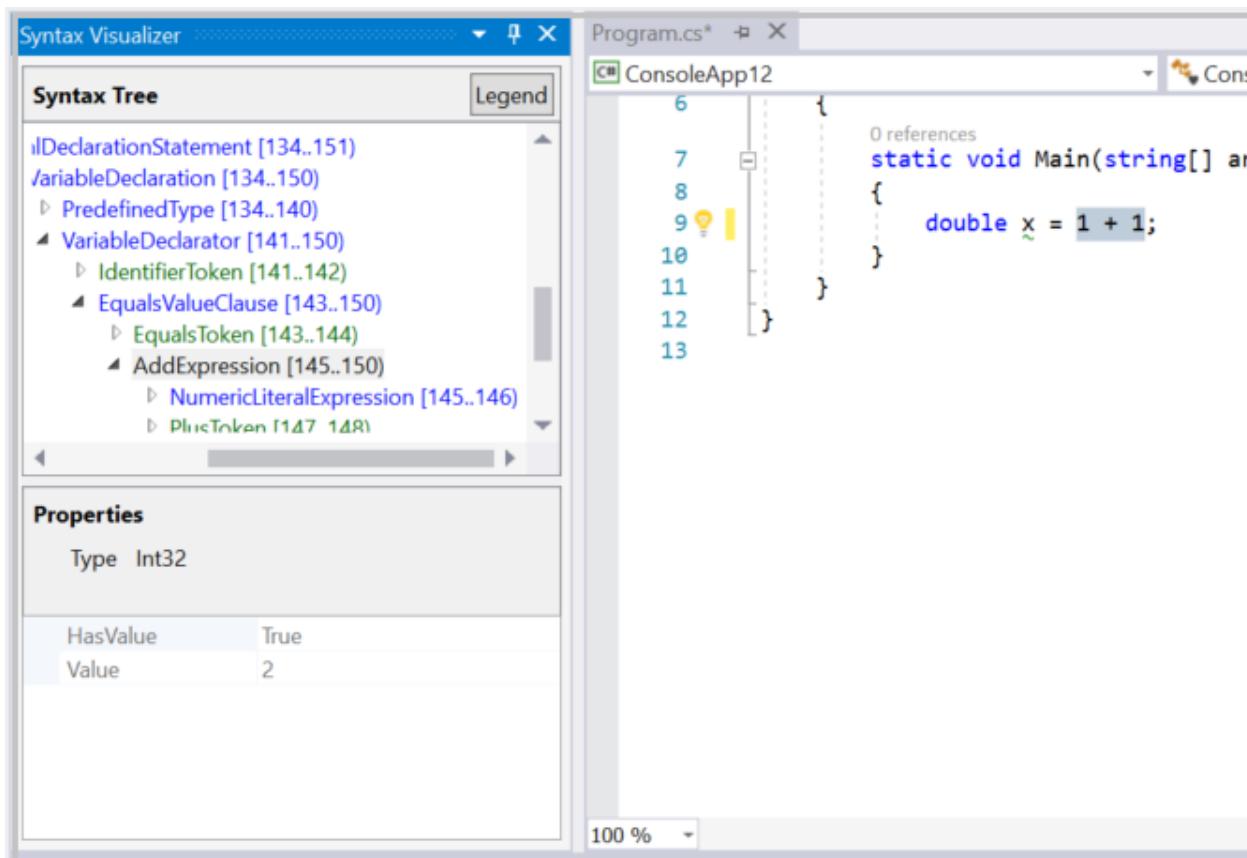
Red arrows point from the text "Intente View Converted TypeSymbol (if any) [Ver TypeSymbol convertido (si existe)] para el mismo nodo AddExpression." to the "Type" and "Name" fields in the Properties grid.

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 cuadricula 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:

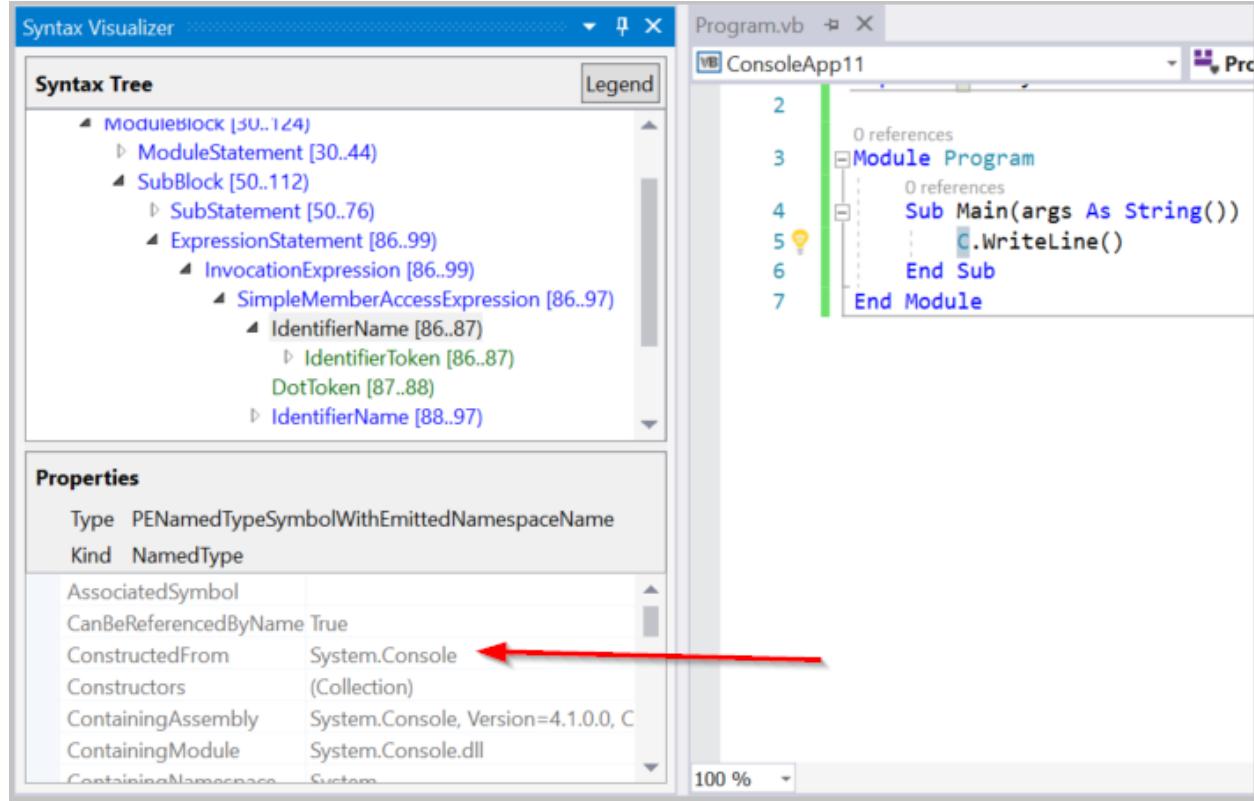
```
VB

Imports C = System.Console

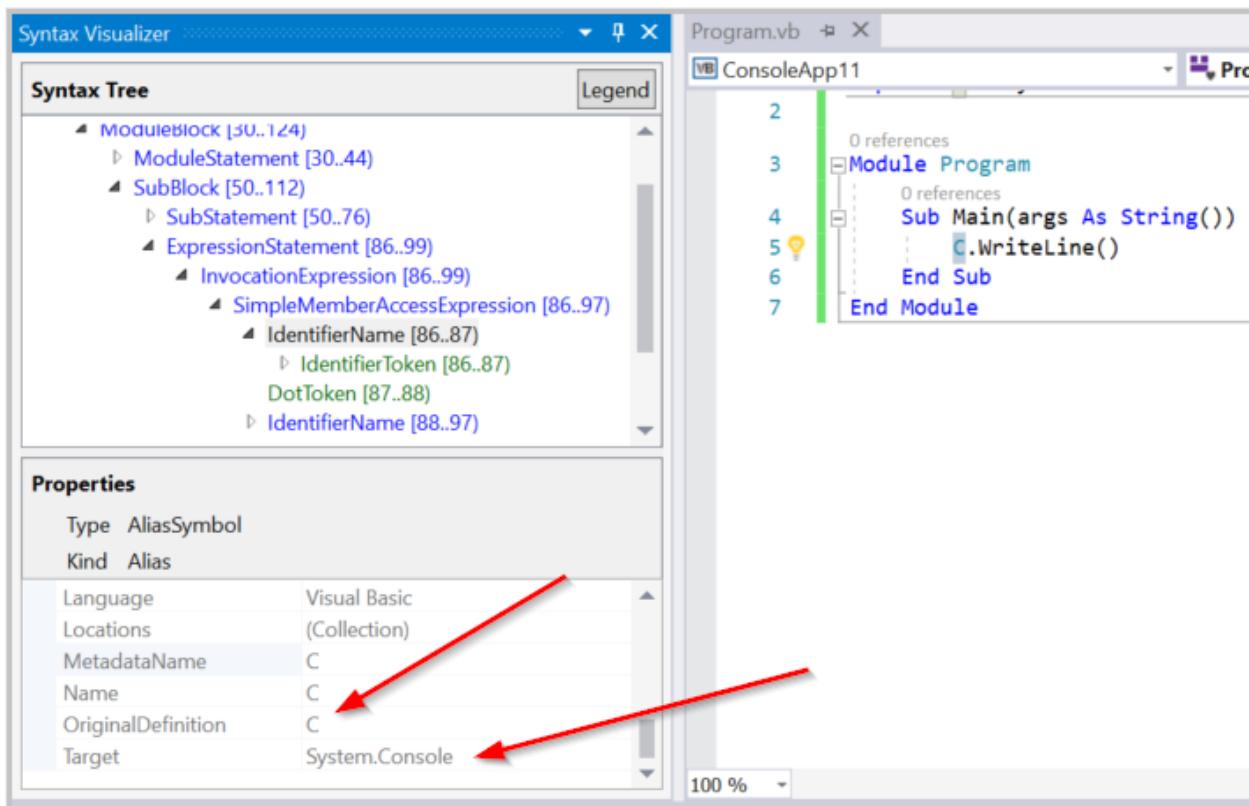
Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

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

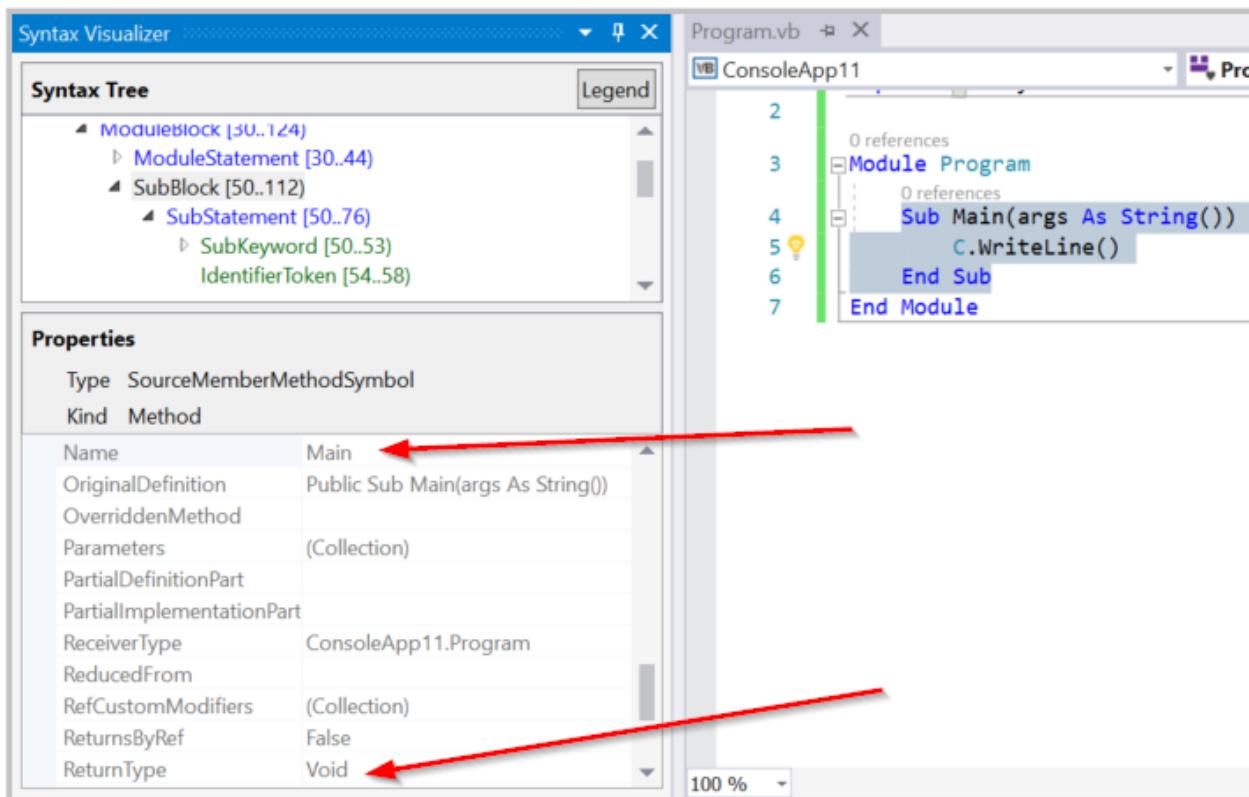
propiedades indica que este identificador está enlazado al tipo `System.Console` tal como se muestra en esta imagen:



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 cuadricula 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.

Elegir identificadores de diagnóstico

Artículo • 21/12/2023

Un identificador de diagnóstico es la cadena asociada a un diagnóstico determinado, como un error del compilador o un diagnóstico generado por un analizador.

Los identificadores se obtienen desde varias API, como:

- [DiagnosticDescriptor.Id](#)
- [ObsoleteAttribute.DiagnosticId](#)
- [ExperimentalAttribute.DiagnosticId](#)

Los identificadores de diagnóstico también se usan como identificadores en el código fuente, por ejemplo, de [#pragma warning disable](#) o archivos [.editorconfig](#).

Consideraciones

- Los identificadores de diagnóstico deben ser únicos
- Los identificadores de diagnóstico deben ser identificadores legales en C#
- Los identificadores de diagnóstico deben tener menos de 15 caracteres de longitud
- Los identificadores de diagnóstico deben tener el formato <PREFIX><number>
 - El prefijo es específico del proyecto
 - El número representa el diagnóstico específico

ⓘ Nota

Cambiar los identificadores de diagnóstico es un cambio importante en el código fuente, ya que las supresiones existentes se omitirán si el identificador ha cambiado.

No limite el prefijo a dos caracteres (como `csxxx` y `CAXXXX`). En su lugar, use un prefijo más largo para evitar conflictos. Por ejemplo, los diagnósticos de `System.*` usan `SYSLIB` como prefijo.



Colaborar con nosotros en
GitHub



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Introducción al análisis de sintaxis

Artículo • 10/05/2023

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”:

```
C#  
  
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 más bajos: la cadena "Hello World!" (Hola mundo) es un **token 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. Selección de **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 interesaría 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. Selección de **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:

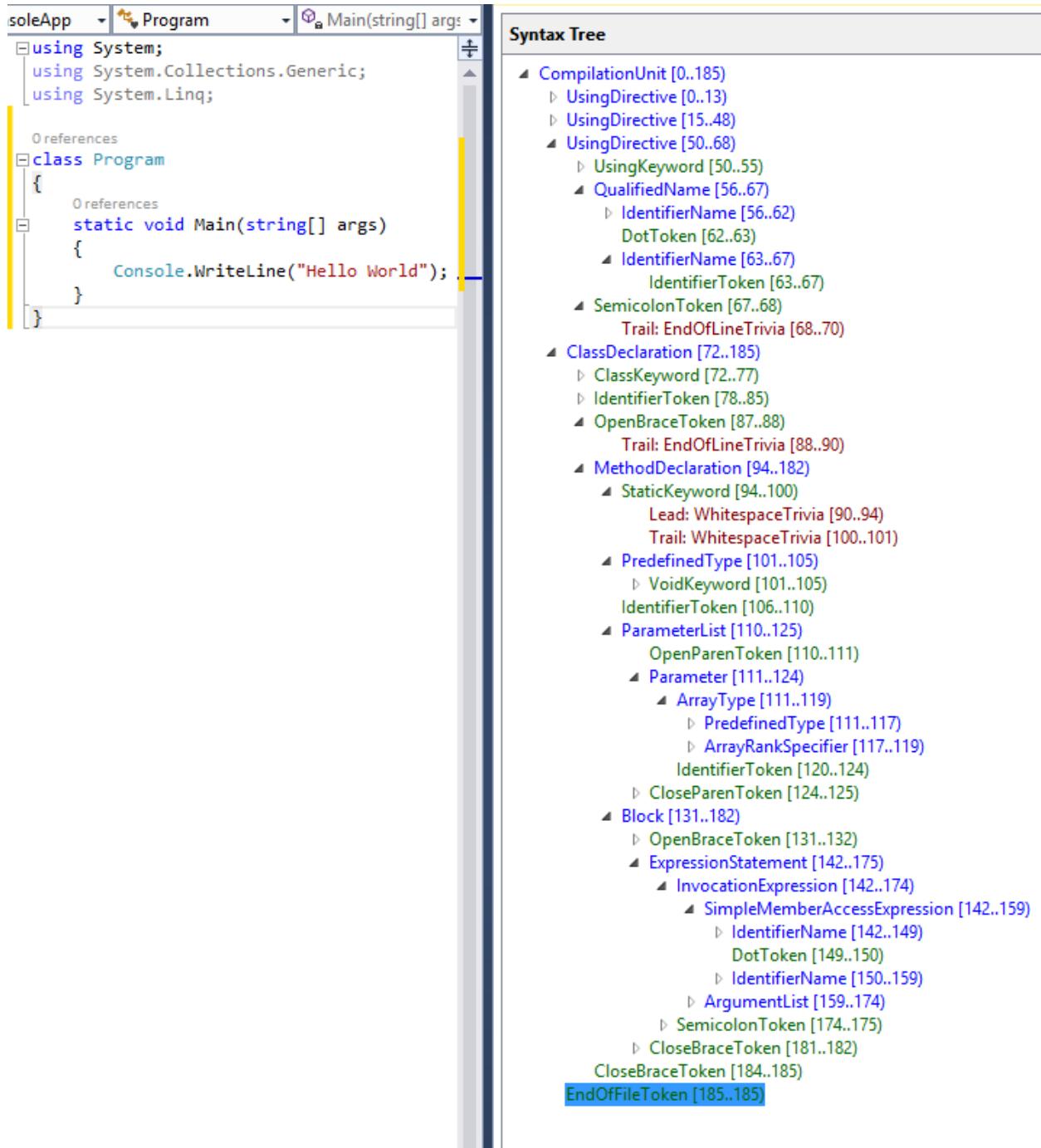
- La clase [Microsoft.CodeAnalysis.SyntaxTree](#), una instancia de lo que representa un árbol de análisis completo. [SyntaxTree](#) es una clase abstracta que tiene derivados específicos del lenguaje. Use los métodos de análisis de la clase [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree](#) (o

`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 preprocessamiento 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](#).

ⓘ Nota

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`:

C#

```
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`:

C#

```
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:

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"{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`:

C#

```
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`:

C#

```
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](#).

C#

```
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:

C#

```
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:

text

```
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`:

C#

```
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:

```
C#  
  
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:

C#

```
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](#).

C#

```
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:

C#

```
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)`:

C#

```
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:

```
C#  
  
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:

Consola

```
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

Artículo • 10/05/2023

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. Selección de **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 interesaría 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. Selección de **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](#).

ⓘ Nota

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**:

C#

```
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`:

C#

```
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:

C#

```
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:

C#

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Enlazar un nombre

La [Compilation](#) crea el [SemanticModel](#) desde el [SyntaxTree](#). Después de crear el modelo, puede consultararlo 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`:

C#

```
// 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`:

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;  
if (systemSymbol?.GetNamespaceMembers() is not null)  
{  
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
```

```
{  
    Console.WriteLine(ns);  
}  
}
```

Ejecute el programa y debería ver la siguiente salida:

Resultados

```
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 . . .
```

ⓘ Nota

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 "Hola, ¡mundo!" que se muestra en la consola.

Para encontrar la cadena "Hola, ¡mundo!", busque 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`:

```
C#  
  
// 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:

```
C#  
  
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`:

```
C#  
  
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](#):

```
C#  
  
var methods = allMembers?.OfType<IMethodSymbol>();
```

A continuación, agregue otro filtro para devolver solo aquellos métodos que son públicos y devuelven una `string`:

```
C#
```

```
var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Seleccione solo la propiedad name y solo los nombres distintos; para ello, elimine las sobrecargas:

C#

```
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:

C#

```
foreach (string name in (from method in stringTypeSymbol?
    .GetMembers().OfType<IMethodSymbol>()
    where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol) &&
    method.DeclaredAccessibility ==
    Accessibility.Public
    select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Compile y ejecute el programa. Debería aparecer la siguiente salida:

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

Artículo • 09/05/2023

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. Selección de **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 interesaría 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. Selección de **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. A continuación, transformará el programa existente reemplazando 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`.

```
C#  
  
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;  
using static System.Console;
```

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`:

C#

```
NameSyntax name = IdentifierName("System");
WriteLine($"\\tCreated the identifier {name}");
```

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`:

C#

```
name = QualifiedName(name, IdentifierName("Collections"));
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`:

C#

```
name = QualifiedName(name, IdentifierName("Generic"));
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`:

```
C#  
  
    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!");  
        }  
    }  
};
```

ⓘ Nota

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:

C#

```
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`:

C#

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
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:

C#

```
root = root.ReplaceNode(oldUsing, newUsing);
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](#) dondequiero 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>Aregar clase...**. En el cuadro de diálogo **Aregar nuevo elemento**, escriba `TypeInferenceRewriter.cs` como nombre de archivo.

Agregue las siguientes directivas using al archivo `TypeInferenceRewriter.cs`:

```
C#  
  
using Microsoft.CodeAnalysis;  
using Microsoft.CodeAnalysis.CSharp;  
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

A continuación, haga que la clase `TypeInferenceRewriter` se extienda a la clase [CSharpSyntaxRewriter](#):

```
C#  
  
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:

```
C#  
  
private readonly SemanticModel SemanticModel;  
  
public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel =  
semanticModel;
```

Invalide el método [VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#):

C#

```
public override SyntaxNode  
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)  
{  
}  
}
```

ⓘ Nota

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 vez 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:

C#

```
Type variable = expression;
```

Si desea explorar por su cuenta, considere la posibilidad de extender el ejemplo finalizado para estos tipos de declaraciones de variable:

C#

```
// 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:

C#

```
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:

C#

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Ahora, agregue esta instrucción para enlazar la expresión de inicializador:

C#

```
var 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:

C#

```
if (SymbolEqualityComparer.Default.Equals(variableType,
    initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
```

```
    }
    else
    {
        return node;
    }
```

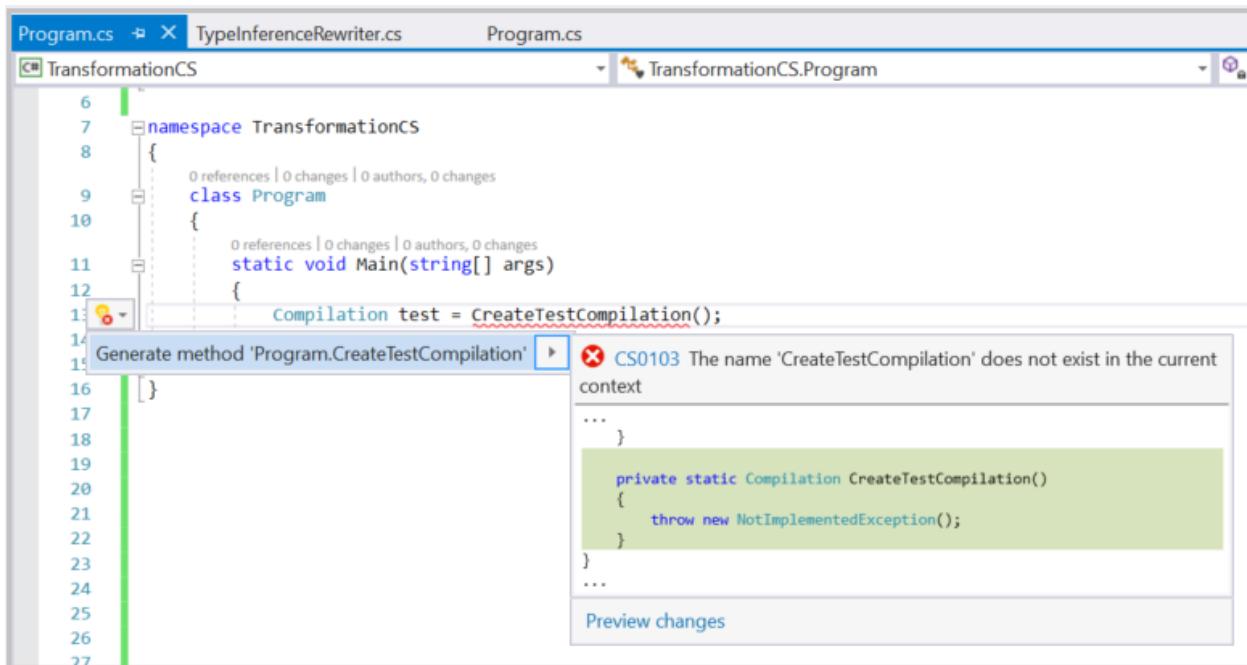
El atributo condicional es necesario porque la declaración puede convertir la expresión de inicializador en una interfaz o clase base. Si se desea, los tipos del lado izquierdo y derecho de la asignación no coinciden. El hecho de quitar el tipo explícito en estos casos también cambiaría la semántica 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:

C#

```
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:

```
C#
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:

```
C#
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
```

```
{  
    File.WriteAllText(sourceTree.FilePath, newSource.ToString());  
}
```

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:

C#

```
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 mscorelib =  
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);  
MetadataReference codeAnalysis =  
    MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);  
MetadataReference csharpCodeAnalysis =  
    MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location)  
;  
  
MetadataReference[] references = { mscorelib, 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

Artículo • 10/05/2023

El SDK de .NET Compiler Platform proporciona las herramientas que necesita para crear diagnósticos personalizados (analizadores), correcciones de código, refactorización de código y supresores de diagnóstico que tengan como destino código de C# o Visual Basic. Un **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, llenar 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, se puede asociar una corrección de código al analizador para representar una modificación en el 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 2019](#), versión 16.8 o posterior

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. Selección de **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. Selección de **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.

Creación de la solución

- 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.

ⓘ Nota

Puede obtener un error de compilación (*MSB4062: No se pudo cargar la tarea "CompareBuildTaskVersion"*). Para corregirlo, actualice los paquetes NuGet de la solución con el Administrador de paquetes NuGet o use `Update-Package` en la ventana Consola del administrador de paquetes.

Exploración de la plantilla del analizador

La plantilla de analizador con corrección de código crea cinco proyectos:

- **MakeConst**, que contiene el analizador.
- **MakeConst.CodeFixes**, que contiene la corrección del código.
- **MakeConst.Package**, que se usa a fin de generar el paquete NuGet para el analizador y la corrección de código.
- **MakeConst.Test**, que es un proyecto de prueba unitaria.
- **MakeConst.Vsix**, que es el proyecto de inicio predeterminado que inicia una segunda instancia de Visual Studio que ha cargado el nuevo analizador. Presione para iniciar el proyecto de VSIX.

ⓘ Nota

Los analizadores deben tener como destino .NET Standard 2.0 porque se pueden ejecutar en el entorno de .NET Core (compilaciones de línea de comandos) y el de .NET Framework (Visual Studio).

💡 Sugerencia

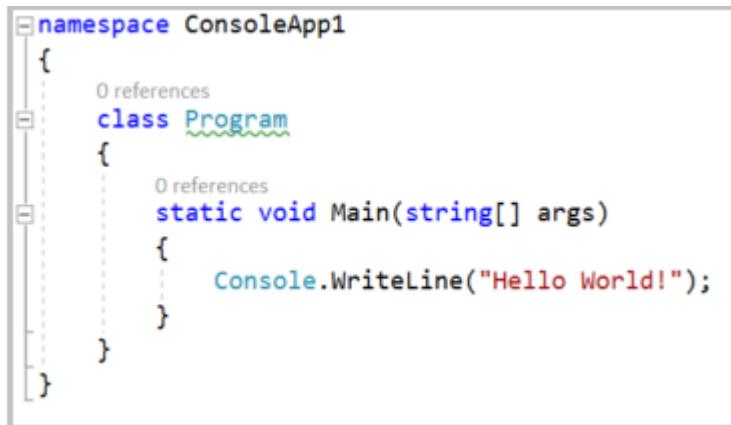
Al ejecutar el analizador, inicie una segunda copia de Visual Studio. Esta segunda copia usa un subárbol del registro diferente para almacenar la configuración. Le permite diferenciar la configuración visual en las dos copias de Visual Studio. Puede

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.

El subárbol incluye no solo el analizador en desarrollo, sino también los analizadores anteriores abiertos. Para restablecer el subárbol Roslyn, debe eliminarlo manualmente de %LocalAppData%\Microsoft\VisualStudio. El nombre de carpeta del subárbol Roslyn terminará en `Roslyn`; por ejemplo, `16.0_9ae182f9Roslyn`. Tenga en cuenta que es posible que tenga que limpiar la solución y volver a generarla después de eliminar el subárbol.

En la segunda instancia de Visual Studio que acaba de iniciar, cree un proyecto de aplicación de consola de C# (servirá cualquier marco; los analizadores funcionan en el 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.

💡 Sugerencia

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.

En este tutorial, se escribe un analizador que notifica al usuario las declaraciones de variables locales que se pueden convertir en constantes locales. Por ejemplo, considere el siguiente código:

C#

```
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`:

C#

```
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.

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 (directa o indirectamente) 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 `AnalyzerDescription` a "Variables that are not modified should be made constants."
- Cambie `AnalyzerMessageFormat` a "Variable '{0}' can be made constant".
- Cambie `AnalyzerTitle` a "Variable can be made constant".

Cuando haya terminado, el editor de recursos debe aparecer tal y como se muestra en la ilustración siguiente:

	Name	Value	Comment
	<code>AnalyzerDescription</code>	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	<code>AnalyzerMessageFormat</code>	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	<code>AnalyzerTitle</code>	Variable can be made constant	The title 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 `MakeConstAnalyzer.Analyzer.Initialize`, busque la línea que registra la acción en los símbolos:

```
C#
```

```
context.RegisterSymbolAction>AnalyzeSymbol, SymbolKind.NamedType);
```

Reemplácela por la línea siguiente:

```
C#
```

```
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:

```
C#
```

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)  
{  
}
```

Cambie `Category` a "Usage" en `MakeConstAnalyzer.cs`, como se muestra en el código siguiente:

```
C#
```

```
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:

```
C#
```

```
int x = 0;
Console.WriteLine(x);
```

El primer paso es encontrar las declaraciones locales. Agregue el código siguiente a `AnalyzeNode` en `MakeConstAnalyzer.cs`:

C#

```
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:

C#

```
// 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`:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis 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.

VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
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`:

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

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:

C#

```
int x = 0;
Console.WriteLine(x);
```

Debe aparecer la bombilla, y el analizador debe informar de un diagnóstico. Sin embargo, en función de la versión de Visual Studio, aparecerá una de estas opciones:

- La bombilla, que todavía usa la corrección de código generada en plantilla, indica que se puede convertir en mayúsculas.
- Un mensaje de banner en la parte superior del editor, que indica que "MakeConstCodeFixProvider" ha encontrado un error y se ha deshabilitado. Esto se debe a que el proveedor de corrección de código aún no se ha cambiado y todavía espera encontrar los elementos `TypeDeclarationSyntax` en vez de los elementos `LocalDeclarationStatementSyntax`.

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:

```
diff
```

```
- int x = 0;  
+ 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 *CodeFixResources.resx* y cambie `CodeFixTitle` a "Make constant".

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.

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:

```
C#
```

```
var declaration =  
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalD  
eclarationStatementSyntax>().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:

```
C#
```

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    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:

C#

```
private static 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`:

C#

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal =
localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

A continuación, agregue el token `const` a la declaración con el siguiente código:

C#

```
// Insert the const token into the modifiers list, creating a new modifiers
list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax 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:

C#

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Se requiere un nuevo espacio de nombres para este código. Agregue la siguiente directiva `using` al principio del archivo:

C#

```
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`:

C#

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode 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 = 2i;
    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.

Compilación de pruebas unitarias

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.

La plantilla usa los paquetes [Microsoft.CodeAnalysis.Testing](#) para las pruebas unitarias.

💡 Sugerencia

La biblioteca de pruebas admite una sintaxis de marcado especial, que incluye lo siguiente:

- [|text|]: indica que se ha notificado un diagnóstico para `text`. De forma predeterminada, este formulario solo se puede usar para probar analizadores con exactamente una instancia de `DiagnosticDescriptor` proporcionada por `DiagnosticAnalyzer.SupportedDiagnostics`.
- {|ExpectedDiagnosticId:text|}: indica que se ha notificado un diagnóstico para `text` con `Id` `ExpectedDiagnosticId`.

Reemplace las pruebas de plantilla de la clase `MakeConstUnitTest` por el método de prueba siguiente:

C#

```
[TestMethod]
public async Task LocalIntCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@""
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
}
```

Ejecute esta prueba para asegurarse de que se supera. En Visual Studio, abra el **Explorador de pruebas**; para ello, seleccione **Prueba>Windows>Explorador de pruebas**. Luego, seleccione **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 el siguiente método de prueba para representar ese caso:

C#

```
[TestMethod]
public async Task VariableIsAssigned_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
"));
}
```

Esta prueba también pasa. Después, agregue métodos de prueba para las condiciones que todavía no ha controlado:

- Declaraciones que ya son `const`, porque ya son constantes:

C#

```
[TestMethod]
public async Task VariableIsAlreadyConst_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
"));
}
```

- Declaraciones que no tienen inicializador, porque no hay ningún valor para usar:

C#

```
[TestMethod]
public async Task NoInitializer_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
"));
}
```

- Declaraciones donde el inicializador no es una constante, porque no pueden ser constantes en tiempo de compilación:

C#

```
[TestMethod]
public async Task InitializerIsNotConstant_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        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:

C#

```
[TestMethod]
public async Task MultipleInitializers_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(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.

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:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis 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.

VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

por el siguiente fragmento de código:

C#

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax 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.

    ISymbol variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    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 de forma adecuada una declaración incorrecta, como `int i = "abc"`, en una constante local. Agregue un método de prueba para este caso:

C#

```
[TestMethod]
public async Task DeclarationIsInvalid_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""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 el 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:

C#

```
[TestMethod]
public async Task DeclarationIsNotString_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        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:

C#

```
[TestMethod]
public async Task StringCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
},
",
@"
using System;

class Program
{
    static void Main()
    {
        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:

```
C#  
  
[TestMethod]  
public async Task VarIntDeclarationCouldBeConstant_Diagnostic()  
{  
    await VerifyCS.VerifyCodeFixAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        [|var item = 4;|]  
    }  
}  
", @"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        const int item = 4;  
    }  
}  
");  
}  
  
[TestMethod]  
public async Task VarStringDeclarationCouldBeConstant_Diagnostic()  
{  
    await VerifyCS.VerifyCodeFixAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        [|var item = ""abc"";|]  
    }  
}  
", @"  
using System;  
  
class Program  
{  
    static void Main()  
    {
```

```
        const string item = ""abc"";  
    }  
}  
");  
}
```

Afortunadamente, todos los errores anteriores se pueden tratar con las mismas técnicas que acaba de aprender.

Para corregir el primer error, abra primero `MakeConstAnalyzer.cs` y busque el bucle `foreach` donde se comprueban todos los inicializadores de la declaración local para asegurarse de que se les hayan asignado 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:

C#

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;  
ITypeSymbol variableType =  
context.SemanticModel.GetTypeInfo(variableTypeName,  
context.CancellationToken).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:

C#

```
// Ensure that the initializer value can be converted to the type of the  
// local declaration without a user-defined conversion.  
Conversion 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.

C#

```
// 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 `MakeConstCodeFixProvider.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`:

C#

```

// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration =
localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another
    // type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.

```

```

        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).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.
            TypeSyntax typeName =
SyntaxFactory.ParseTypeName(type.ToString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration =
variableDeclaration.WithType(simplifiedTypeName);
        }
    }

// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal =
trimmedLocal.WithModifiers(newModifiers)
    .WithDeclaration(variableDeclaration);

```

Deberá agregar una directiva `using` para usar el tipo `Simplifier`:

C#

```
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 trata correctamente.

Por último, agregue el código siguiente:

```
C#
```

```
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](#)

Conceptos de programación (C#)

Artículo • 11/04/2024

En esta sección se explican los conceptos de programación del lenguaje C#.

En esta sección

 Expandir tabla

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 <code>Async</code> y <code>Await</code> 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.
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.

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto ↗](#)

Covarianza y contravarianza (C#)

Artículo • 11/04/2024

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.

```
C#  
  
// 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.

```
C#  
  
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.

C#

```
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.

C#

```
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

 Expandir tabla

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.
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 <code>IEnumerable<T></code> y <code>IComparable<T></code> 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.
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 <code>Func</code> y <code>Action</code> puede ayudarle a volver a usar el código.

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto ↗](#)

Varianza en interfaces genéricas (C#)

Artículo • 09/05/2023

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.

C#

```
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.

C#

```
// 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.

C#

```
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.

C#

```
// 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#)

Artículo • 09/05/2023

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*.

ⓘ Nota

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.

ⓘ Importante

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.

C#

```
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 \(C#\)](#).

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.

C#

```
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.

C#

```
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.

C#

```
interface IInvariant<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.

C#

```
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:

C#

```
// 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.

C#

```
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.

C#

```
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.

C#

```
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.

C#

```
// 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#)

Artículo • 10/05/2023

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`.

C#

```
// 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 `IEqualityComparer<T>`. La clase `PersonComparer` implementa la interfaz `IEqualityComparer<Person>`. Pero se puede volver a usar esta clase para comparar una secuencia de objetos del tipo `Employee` porque `Employee` hereda `Person`.

C#

```

// 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());
    }

}

```

Vea también

- [Varianza en interfaces genéricas \(C#\)](#)

Varianza en delegados (C#)

Artículo • 10/05/2023

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.

C#

```
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.

C#

```
// 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.

C#

```
// 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.

C#

```
// 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`.

C#

```
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.

C#

```
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.

C#

```
public delegate void DContravariant<in A>(A a);
```

ⓘ Importante

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.

C#

```
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.

C#

```
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.

C#

```
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.

C#

```
// 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#)

Artículo • 07/04/2023

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

Descripción

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

C#

```
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

Descripción

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:

C#

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Un delegado [MouseEventHandler](#) que define la firma del evento [Button.MouseClick](#). Su firma es:

C#

```
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

C#

```
// 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;

}
```

Consulte 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#)

Artículo • 07/04/2023

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`.

C#

```
// 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`.

C#

```
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

Iteradores (C#)

Artículo • 10/05/2023

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.

```
C#  
  
static void Main()  
{  
    foreach (int number in SomeNumbers())  
    {  
        Console.WriteLine(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.

ⓘ Nota

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.

C#

```
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`.

```
C#  
  
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`.

```
C#  
  
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 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`.

C#

```

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)
{
    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--)
        {
            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 llenar 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](#)
- [Utilizar foreach con matrices](#)
- [Genéricos](#)

Instrucciones (Guía de programación de C#)

Artículo • 07/04/2023

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:

```
C#  
  
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.
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, vea los temas siguientes: <ul style="list-style-type: none"> • if • switch
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, vea los temas siguientes: <ul style="list-style-type: none"> • do • for • foreach • while

Categoría	Palabras clave de C# / notas
Instrucciones de salto	<p>Las instrucciones de salto transfieren el control a otra sección de código. Para obtener más información, vea los temas siguientes:</p> <ul style="list-style-type: none"> • break • continue • goto • return • yield
Instrucciones para el control de excepciones	<p>Las instrucciones para el control de excepciones permiten recuperarse correctamente de condiciones excepcionales producidas en tiempo de ejecución. Para obtener más información, vea los temas siguientes:</p> <ul style="list-style-type: none"> • throw • try-catch • try-finally • try-catch-finally
checked y unchecked	<p>Las instrucciones <code>checked</code> y <code>unchecked</code> permiten especificar si las operaciones numéricas de tipo entero pueden producir un desbordamiento cuando el resultado se almacena en una variable que es demasiado pequeña para contener el valor resultante.</p>
Instrucción <code>await</code>	<p>Si marca un método con el modificador async, puede usar el operador await en el método. Cuando el control alcanza una expresión <code>await</code> 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.</p> <p>Para obtener un ejemplo sencillo, vea la sección "Métodos asincrónicos" de Métodos. Para obtener más información, vea Programación asincrónica con Async y Await.</p>
Instrucción <code>yield</code> <code>return</code>	<p>Un iterador realiza una iteración personalizada en una colección, como una lista o matriz. Un iterador utiliza la instrucción yield return para devolver cada elemento de uno en uno. Cuando se alcanza una instrucción <code>yield return</code>, se recuerda la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama el iterador.</p> <p>Para obtener más información, consulta Iteradores.</p>
Instrucción <code>fixed</code>	<p>La instrucción <code>fixed</code> impide que el recolector de elementos no utilizados cambie la ubicación de una variable móvil. Para obtener más información, vea fixed.</p>
Instrucción <code>lock</code>	<p>La instrucción <code>lock</code> permite limitar el acceso a bloques de código a un solo subproceso de cada vez. Para obtener más información, vea lock.</p>

Categoría	Palabras clave de C# / notas
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.

```
C#
// 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.

```
C#
// 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:

```
C#  
  
void ProcessMessages()  
{  
    while (ProcessMessage())  
        ; // Statement needed here.  
}  
  
void F()  
{  
    //...  
    if (done) goto exit;  
//...  
exit:  
    ; // Statement needed here.  
}
```

Instrucciones insertadas

Algunas instrucciones, por ejemplo, las [instrucciones de iteración](#), 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:

```
C#  
  
// 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:

```
C#
```

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Coloque la instrucción insertada en un bloque para solucionar el error:

```
C#
```

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToString());
}
```

Bloques de instrucciones anidadas

Los bloques de instrucciones pueden anidarse, como se muestra en el código siguiente:

```
C#
```

```
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:

```
C#
```

```
// 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#](#).

Consulte también

- [Guía de programación de C#](#)
- [Palabras clave de instrucciones](#)
- [Operadores y expresiones de C#](#)

Miembros con cuerpo de expresión (Guía de programación de C#)

Artículo • 16/08/2023

Las definiciones de cuerpos de expresión permiten proporcionar la implementación de un miembro de una forma 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:

```
C#  
  
member => expression;
```

donde *expresión* es una expresión válida.

Las definiciones del cuerpo de expresiones se pueden usar con los miembros de tipo siguientes:

- [Método](#)
- [Propiedad de solo lectura](#)
- [Propiedad](#)
- [Constructor](#)
- [Finalizador](#)
- [Indizador](#)

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. La palabra clave `return` no se usa en la definición de cuerpo de expresión de `ToString`.

```
C#
```

```

using System;

namespace ExpressionBodiedMembers;

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

Puede usar la definición del cuerpo de expresiones para implementar una propiedad de solo lectura. Para ello, use la sintaxis siguiente:

C#

```
PropertyTypePropertyName => 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`:

C#

```
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

Puede usar las definiciones del cuerpo de expresiones para implementar los descriptores de acceso `get` y `set` de propiedades. En el ejemplo siguiente se muestra cómo hacerlo:

C#

```
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`.

C#

```
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.

C#

```
public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer 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 algunos deportes. Los descriptores de acceso `get` y `set` del indizador se implementan como definiciones de cuerpos de expresión.

C#

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

namespace SportsExample;

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 \(Guía de programación de C#\)](#).

Consulte también

- [Reglas de estilo de código de .NET para miembros con forma de expresión](#)

Comparaciones de igualdad (guía de programación de C#)

Artículo • 13/03/2024

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.

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    public 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);
    }
}
```

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.

```
C#  
  
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 el caso de los registros, la igualdad de valores significa que dos variables de un tipo de registro son iguales si los tipos coinciden y todos los valores de propiedad y campo coinciden.

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

 Expandir tabla

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.
Tipos	Proporciona información sobre el sistema de tipos de C# y vínculos a información adicional.
Registros	Proporciona información sobre los tipos de registro, que comprueban la igualdad de valores de forma predeterminada.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

nuestra guía para
colaboradores.

Definición de la igualdad de valores para una clase o una estructura (Guía de programación de C#)

Artículo • 07/04/2023

Los [registros](#) implementan automáticamente la igualdad de valores. Considere la posibilidad de definir `record` en lugar de `class` cuando el tipo modela los datos y debe implementar la igualdad de valores.

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, 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. La propiedad reflexiva `x.Equals(x)` devuelve `true`.
2. La propiedad simétrica `x.Equals(y)` devuelve el mismo valor que `y.Equals(x)`.
3. La propiedad transitiva: si `(x.Equals(y) && y.Equals(z))` devuelve `true`,
`x.Equals(z)` devuelve `true`.
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. Sin embargo, `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 `Object.Equals(Object)virtual`. 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 comparan uno o dos campos del tipo. No genere excepciones desde `Equals`. Para las clases que están relacionadas por herencia:
 - 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 llame a `base.Equals` si el tipo hereda directamente de `Object`, porque la implementación `Object` de `Object.Equals(Object)` realiza una comprobación de igualdad de referencia).
 - Dos variables deben considerarse iguales solo si los tipos en tiempo de ejecución de las variables que se van a comparar son los mismos. Además, asegúrese de que se utiliza la implementación `IEquatable` del método `Equals` para el tipo en tiempo de ejecución si los tipos en tiempo de ejecución y en tiempo de compilación de una variable son diferentes. Una estrategia para asegurarse de que los tipos en tiempo de ejecución siempre se comparan correctamente es implementar `IEquatable` solo en clases `sealed`. Para obtener más información, vea el [ejemplo de clases](#) más adelante en este artículo.
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 `>=`.

 **Nota**

A partir de C# 9.0, puede usar registros para obtener la semántica de igualdad de valores sin código reutilizable innecesario.

Ejemplo de clase

En el ejemplo siguiente se muestra cómo implementar la igualdad de valores en una clase (tipo de referencia).

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object obj) => this.Equals(obj as
TwoDPoint);

    public bool Equals(TwoDPoint p)
    {
        if (p is 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() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                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) => !(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 ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as
ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is 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() => (X, Y, Z).GetHashCode();

public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
{
    if (lhs is null)
    {
        if (rhs is 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) => !(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
   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.

ⓘ Importante

Es posible que el código de ejemplo anterior no controle cada escenario de herencia de la manera esperada. Observe el código siguiente:

C#

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

Este código notifica que `p1` es igual a `p2` pesar de la diferencia en los valores `z`. La diferencia se omite porque el compilador elige la implementación `TwoDPoint` de `IEquatable` basándose en el tipo en tiempo de compilación.

La igualdad de valores integrada de los tipos `record` controla escenarios como este. Si `TwoDPoint` y `ThreeDPoint` fueran de tipo `record`, el resultado de `p1.Equals(p2)` sería `False`. Para obtener más información, vea [Igualdad en las jerarquías de herencia de tipo record](#).

Ejemplo de estructura

En el ejemplo siguiente se muestra cómo implementar la igualdad de valores en un `struct` (tipo de valor):

C#

```
namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other
&& this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();
    }
}
```

```

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !
(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        int i = 5;

        // 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);

        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 reemplaza el método `Equals` virtual en una estructura, 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 de la estructura.

Los operadores `==` y `!=` no pueden funcionar en un struct a menos que el struct los sobrecargue explícitamente.

Consulte también

- [Comparaciones de igualdad](#)
- [Guía de programación de C#](#)

Procedimiento Probar la igualdad de referencias (identidad) (Guía de programación de C#)

Artículo • 07/04/2023

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

```
C#  
  
using System.Text;  
  
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 assignment: 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.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 assignment: 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 assignment: ReferenceEquals(tcA, tcB) = True
 * tcB.Name = TestClass 42 tcB.Num: 42
 * After assignment: ReferenceEquals(tsC, tsD) = False
 * ReferenceEquals(strA, strB) = True
 * strA = "Goodbye world!" strB = "Hello world!"
 * After strA changes, ReferenceEquals(strA, strB) = False
 * ReferenceEquals(stringC, strB) = False
 * stringC == strB = True
 */

```

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. Pero 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.

Consulte también

- [Comparaciones de igualdad](#)

Conversiones de tipos (Guía de programación de C#)

Artículo • 15/03/2024

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:

C#

```
int i;  
  
// error CS0029: can't implicitly convert type 'string' to 'int'  
i = "Hello";
```

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 sintaxis especial porque la conversión siempre se realiza correctamente y no se pierde 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 usan métodos especiales que puede definir para habilitar conversiones explícitas e implícitas entre tipos personalizados que no tienen una relación de clase derivada

de clase-base. 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`.

C#

```
// 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.

C#

```
Derived d = new Derived();
```

```
// Always OK.  
Base b = d;
```

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 o la conversión de tipos puede fallar en tiempo de ejecución. 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.

C#

```
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:

C#

```
// 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 tipos que produce un error en tiempo de ejecución hace que se produzca un [InvalidCastException](#).

C#

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "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.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

El método `Test` tiene un parámetro `Animal`, por lo que la conversión explícita del argumento `a` en un `Reptile` supone una suposición peligrosa. Es más seguro no hacer suposiciones, sino comprobar el tipo. 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#](#).

Consulte también

- [Tipos](#)
- [Expresión Cast](#)
- [Operadores de conversión definidos por el usuario](#)
- [Conversión de tipos generalizada](#)
- [Procedimiento Convertir una cadena en un número](#)



Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

[Proporcionar comentarios sobre el producto](#)

Conversión boxing y unboxing (Guía de programación de C#)

Artículo • 07/04/2023

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`.

C#

```
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`:

C#

```
o = 123;
i = (int)o; // unboxing
```

En los ejemplos siguientes se muestra cómo usar la conversión boxing en C#.

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
// 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:

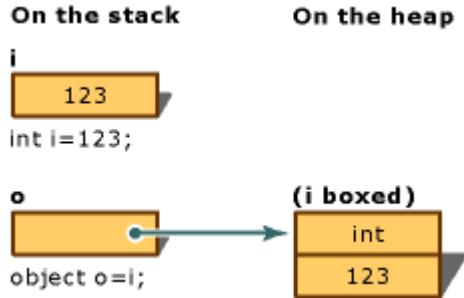
```
C#  
  
int i = 123;
```

La siguiente instrucción aplica implícitamente la operación de conversión boxing en la variable `i`:

```
C#  
  
// 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:

C#

```
int i = 123;
object o = (object)i; // explicit boxing
```

Ejemplo

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.

C#

```
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
```

```
The object-type value = 123
```

```
*/
```

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:

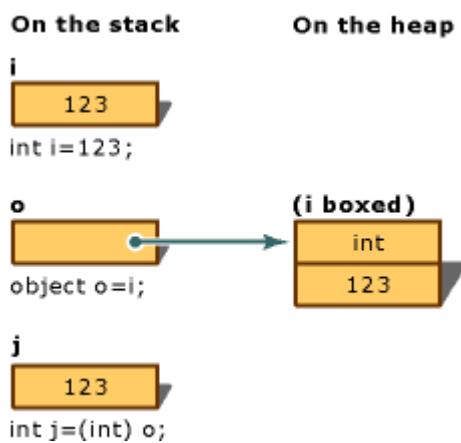
- Comprobar la instancia de objeto para asegurarse de que se trata de un valor de conversión boxing del tipo de valor dado.
- Copiar el valor de la instancia en la variable de tipo de valor.

Las siguientes instrucciones muestran las operaciones de conversión boxing y unboxing:

```
C#
```

```
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.

C#

```
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:

C#

```
int j = (short)o;
```

a:

C#

```
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#)

Artículo • 07/04/2023

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
<code>bool</code>	.ToBoolean(Byte[], Int32)
<code>char</code>	ToChar(Byte[], Int32)
<code>double</code>	.ToDouble(Byte[], Int32)
<code>short</code>	ToInt16(Byte[], Int32)
<code>int</code>	ToInt32(Byte[], Int32)
<code>long</code>	ToInt64(Byte[], Int32)
<code>float</code>	ToSingle(Byte[], Int32)
<code>ushort</code>	ToUInt16(Byte[], Int32)
<code>uint</code>	ToUInt32(Byte[], Int32)
<code>ulong</code>	ToUInt64(Byte[], Int32)

Ejemplos

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.

ⓘ Nota

El resultado puede cambiar en función de los modos endian de la arquitectura del equipo.

C#

```
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
```

En este ejemplo, el método [GetBytes\(Int32\)](#) de la clase [BitConverter](#) se llama para convertir `int` en una matriz de bytes.

ⓘ Nota

El resultado puede cambiar en función de los modos endian de la arquitectura del equipo.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

Consulte también

- [BitConverter](#)
- [IsLittleEndian](#)
- [Tipos](#)

Procedimiento Convertir una cadena en un número (Guía de programación de C#)

Artículo • 07/11/2024

Puede convertir `string` en un número si llama al método `Parse` o `TryParse` que se encuentra en tipos numéricos (`int`, `long`, `double`, etc.), o bien mediante los métodos de la clase [System.Convert](#).

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 contenga la cadena, 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 booleano 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` inicia una excepción, pero `TryParse` devuelve `false`. Cuando se llama a un método `Parse`, siempre debe usar control de excepciones para detectar un [FormatException](#) cuando se produzca un error en la operación de análisis.

Llamada a métodos Parse o TryParse

Los métodos `Parse` y `TryParse` no tienen en cuenta los espacios en blanco al principio ni 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 el método `decimal.TryParse` para analizar "10", "10,3" o " 10 ", pero no para analizar 10 en "10X", "1 0" (fíjese en el espacio insertado), "10 ,3" (fíjese en el espacio insertado) o "10e1" (ahí funciona `float.TryParse`), y así sucesivamente. 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`.

C#

```
using System;

public static 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.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
```

```

        Console.WriteLine(numValue);
    }
    else
    {
        Console.WriteLine($"Int32.TryParse could not parse
'{inputString}' to an int.");
    }
    // Output: Int32.TryParse could not parse 'abc' to an int.
}
}

```

En el ejemplo siguiente se muestra 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 contiene pocos 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.

C#

```

using System;

public static 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 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.

[\[+\] Expandir tabla](#)

Tipo numérico	Método
<code>decimal</code>	ToDecimal(String)
<code>float</code>	ToSingle(String)
<code>double</code>	.ToDouble(String)
<code>short</code>	ToInt16(String)
<code>int</code>	ToInt32(String)
<code>long</code>	ToInt64(String)
<code>ushort</code>	ToUInt16(String)
<code>uint</code>	ToUInt32(String)
<code>ulong</code>	ToUInt64(String)

En este ejemplo, se llama al método `Convert.ToInt32(String)` para convertir una cadena de entrada en un valor `int`. En el ejemplo se detectan las dos excepciones más comunes producidas por este método: `FormatException` y `OverflowException`. Si se puede incrementar el número resultante sin exceder `Int32.MaxValue`, el ejemplo suma 1 al resultado y muestra la salida.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.WriteLine("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
```

Uso de GitHub Copilot para convertir una cadena en un número

Puede usar GitHub Copilot en el IDE para generar código para convertir una cadena en un número en C#.

Si usa [Visual Studio 2022, versión 17.8 o posterior](#), puede probar el Copilot de GitHub controlado por [IA en Visual Studio](#) para generar código para convertir una cadena en un número. Envíe su pregunta como solicitud en la ventana de chat de Copilot, como en el ejemplo siguiente. También puede enviar mensajes mediante el [chat en línea](#) en la propia ventana del editor.

ⓘ Nota

GitHub Copilot funciona con IA, por lo que es posible que se produzcan sorpresas y errores. Asegúrese de comprobar cualquier código o sugerencia generado. Para obtener más información sobre el uso general de GitHub Copilot, impacto en el producto, supervisión humana y privacidad, consulte [Preguntas más frecuentes sobre GitHub Copilot ↗](#).

En el texto siguiente se muestra un mensaje de ejemplo para Copilot Chat:

Genere código de C# para convertir la cadena "123" en un número mediante TryParse. Proporcione una salida de ejemplo.

Puede personalizar la solicitud para usar una cadena según sus requisitos.

Puede usar características de chat, como [comandos de barra diagonal](#), [referencias a archivos](#), [métodos](#), [clases](#) y [subprocesos](#), para establecer la intención y obtener mejores respuestas con contexto con ámbito. Para un archivo existente que está abierto en el IDE, puede preguntar a GitHub Copilot mediante el chat en línea con `/generate code to convert the string string1 in #Filename to a number.`

En la salida siguiente se muestra un ejemplo de respuesta de Copilot Chat:

Output

```
Here is a C# code snippet to convert the string "123" to a number using TryParse:  
  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        string numberString = "123";  
        int number;  
        bool success = int.TryParse(numberString, out number);  
  
        if (success)  
        {  
            // Example output  
            Console.WriteLine("The converted number is: " + number);  
        }  
        else  
        {  
            Console.WriteLine("Conversion failed.");  
        }  
    }  
}
```

When you run this code, the output will be:
The converted number is: 123

Cuando Copilot devuelve un bloque de código, la respuesta incluye opciones para copiar el código, insertar el código en un nuevo archivo u obtener una vista previa de la salida del código.

ⓘ Nota

Los resultados pueden ser diferentes de lo que se muestra en las respuestas de ejemplo. Los modelos de IA no son deterministas, lo que significa que pueden devolver respuestas diferentes cuando se le haga la misma pregunta. Esto puede

deberse al aprendizaje y la adaptación adicionales a lo largo del tiempo, la variación del lenguaje, los cambios en el contexto, como el historial de chat, etc.

GitHub Copilot Chat

Generate C# code to convert the string "123" to a number. Provide an example output.

generate c# code to convert the string "123" to a number using TryParse. Provide example output

GitHub Copilot

Here is a C# code snippet to convert the string "123" to a number using TryParse:

```
using System;

class Program
{
    static void Main()
    {
        string numberString = "123";
        int number;
        bool success = int.TryParse(numberString, out number);

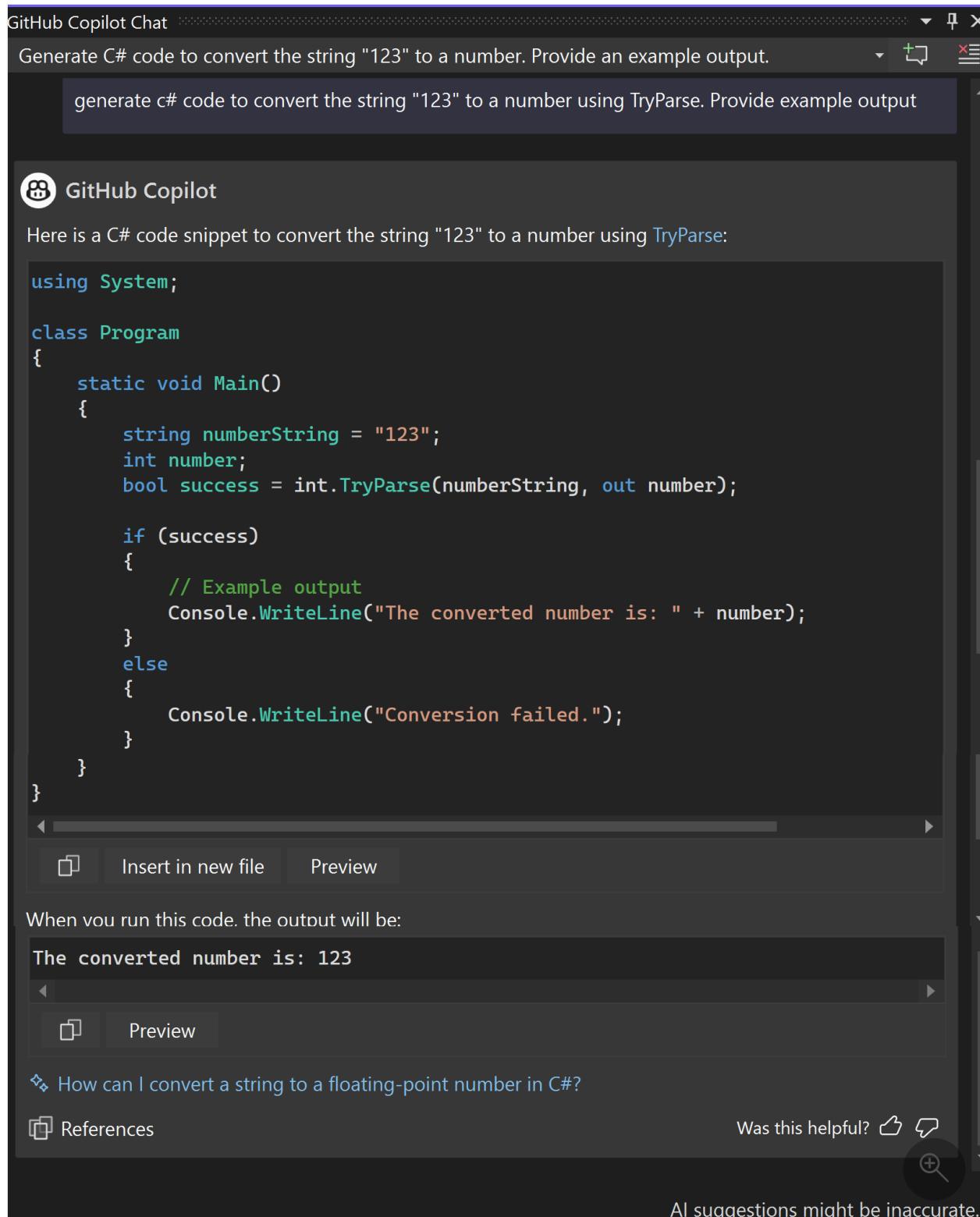
        if (success)
        {
            // Example output
            Console.WriteLine("The converted number is: " + number);
        }
        else
        {
            Console.WriteLine("Conversion failed.");
        }
    }
}
```

When you run this code, the output will be:

The converted number is: 123

Was this helpful?

AI suggestions might be inaccurate.



Para más información, vea:

- [Centro de confianza de GitHub Copilot](#)
- [GitHub Copilot en Visual Studio](#)
- [GitHub Copilot en VS Code](#)

Procedimiento Convertir cadenas hexadecimales en tipos numéricos (Guía de programación de C#)

Artículo • 09/05/2023

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.

Ejemplos

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`.

C#

```
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
*/
```

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`.

C#

```
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 !
*/
```

En este ejemplo se muestra otra manera de convertir un `string` hexadecimal en un entero mediante la llamada al método `Parse(String, NumberStyles)`.

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

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`.

C#

```
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
```

En el ejemplo siguiente se muestra cómo convertir una matriz `byte` en una cadena hexadecimal mediante la clase `System.BitConverter`.

C#

```
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
*/
```

En el ejemplo siguiente se muestra cómo convertir una matriz `byte` en una cadena hexadecimal mediante una llamada al método `Convert.ToString` introducido en .NET 5.0.

C#

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
646F74636574
*/
```

Vea también

- [Cadenas con formato numérico estándar](#)
- [Tipos](#)
- [Determinación de si una cadena representa un valor numérico](#)

Control de versiones con las palabras clave Override y New (Guía de programación de C#)

Artículo • 07/04/2023

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`:

```
C#  
  
class GraphicsClass  
{  
    public virtual void DrawLine() { }  
    public virtual void DrawPoint() { }  
}
```

Su compañía usa esta clase y usted la usa para衍生 su propia clase, agregando un nuevo método:

```
C#  
  
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:

```
C#  
  
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`:

C#

```
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:

C#

```
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:

C#

```
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:

```
C#  
  
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 de 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:

```
C#  
  
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:

```
C#
```

```
((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#](#)
- [El sistema de tipos de C#](#)
- [Métodos](#)
- [Herencia](#)

Saber cuándo utilizar las palabras clave Override y New (Guía de programación de C#)

Artículo • 07/04/2023

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`.

```
C#  
  
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.

C#

```
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`.

C#

```
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.

C#

```
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.

C#

```
// 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`.

C#

```
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`.

C#

```
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`.

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Vuelva a ejecutar el proyecto. Observe especialmente las dos últimas líneas del resultado siguiente.

C#

```
// 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.

C#

```
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`.

C#

```

// 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.

C#

```
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.

C#

```
// 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`.

C#

```
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`.

```
C#  
  
// 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.

```
C#  
  
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.

```
C#
```

```

// 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.

C#

```

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#](#)
- [El sistema de tipos de C#](#)
- [Control de versiones con las palabras clave Override y New](#)
- [base](#)
- [abstract](#)

Procedimiento para invalidar el método `ToString` (Guía de programación de C#)

Artículo • 13/03/2024

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:

C#

```
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](#).

ⓘ Importante

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:

C#

```
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.

```
C#  
  
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:

```
C#  
  
Person person = new Person { Name = "John", Age = 12 };  
Console.WriteLine(person);  
// Output:  
// Person: John 12
```

Consulte también

- [IFormattable](#)
- [El sistema de tipos de C#](#)
- [Cadenas](#)
- [string](#)
- [override](#)
- [virtual](#)
- [Aplicación de formato a tipos](#)



[Colaborar con nosotros en GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

de incorporación de cambios.
Para más información, consulte
[nuestra guía para
colaboradores](#).

 Abrir incidencia con la
documentación

 Proporcionar comentarios sobre
el producto

Miembros (Guía de programación de C#)

Artículo • 07/04/2023

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:

Miembro	Descripción
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.

Miembro	Descripción
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.
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#)

Artículo • 07/04/2023

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:

```
C#  
  
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:

```
C#  
  
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:

```
C#  
  
// 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:

```
C#
```

```
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:

C#

```
public class D : C
{
    public sealed override void DoWork() { }
```

Consulte también

- [Guía de programación de C#](#)
- [El sistema de tipos de C#](#)
- [Herencia](#)
- [Métodos](#)
- [Campos](#)
- [Procedimiento para definir propiedades abstractas](#)

Clases estáticas y sus miembros (Guía de programación de C#)

Artículo • 15/03/2024

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:

C#

```
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.

C#

```
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 tipos 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. No obstante, existe la garantía de que se cargará, sus campos se inicializarán y 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.

ⓘ Nota

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 garantiza 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 o interfaz, 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:

C#

```
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() ?? "0");
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                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 existe 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 o 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`, un campo `const` es básicamente estático en su comportamiento. Pertece 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:

C#

```
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:

C#

```
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. No obstante, la mayoría de las veces la diferencia de rendimiento entre las dos no es significativa.

Especificación del lenguaje C#

Para más información, consulte [Clases estáticas, Miembros estáticos y de instancia y Constructores estáticos](#) en la [especificación del lenguaje C#](#). La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Consulte también

- [static](#)
- [Clases](#)
- [class](#)
- [Constructores estáticos](#)
- [Constructores de instancias](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Modificadores de acceso (Guía de programación de C#)

Artículo • 20/08/2024

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. Un [ensamblado](#) es un archivo `.dll` o `.exe` creado mediante la compilación de uno o varios archivos `.cs` en una sola compilación. Use los modificadores de acceso siguientes para especificar la accesibilidad de un tipo o miembro cuando lo declare:

- [public](#): el código de cualquier ensamblado puede tener acceso a este tipo o miembro. El nivel de accesibilidad del tipo contenedor controla el nivel de accesibilidad de los miembros públicos del tipo.
- [private](#): solo el código declarado en el mismo `class` o `struct` puede acceder a este miembro.
- [protected](#): solo el código en el mismo `class` o en un `class` derivado puede tener acceso a este tipo o miembro.
- [internal](#): solo el código del mismo ensamblado puede tener acceso a este tipo o miembro.
- [protected internal](#): solo el código del mismo ensamblado o de una clase derivada de otro ensamblado puede tener acceso a este tipo o miembro.
- [private protected](#): solo el código del mismo ensamblado y en la misma clase o una clase derivada puede tener acceso al tipo o miembro.
- [archivo](#): solo el código del mismo archivo puede tener acceso al tipo o al miembro.

El modificador [record](#) de un tipo hace que el compilador sintetice miembros adicionales. El modificador `record` no afecta a la accesibilidad predeterminada para `record class` o `record struct`.

Tabla de resumen

 Expandir tabla

Ubicación del autor de la llamada	<code>public</code>	<code>protected</code>	<code>protected internal</code>	<code>internal</code>	<code>private</code>	<code>private protected</code>	<code>file</code>
Dentro del archivo	✓	✓	✓	✓	✓	✓	✓

Ubicación del autor de la llamada	public	protected internal	protected	internal	private protected	private	file
Dentro de la clase	✓	✓	✓	✓	✓	✓	✗
Clase derivada (mismo ensamblado)	✓	✓	✓	✓	✓	✗	✗
Clase no derivada (mismo ensamblado)	✓	✓	✗	✓	✗	✗	✗
Clase derivada (otro ensamblado)	✓	✓	✓	✗	✗	✗	✗
Clase no derivada (otro ensamblado)	✓	✗	✗	✗	✗	✗	✗

En los ejemplos siguientes se muestra cómo especificar modificadores de acceso en un tipo y miembro:

C#

```
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 del tipo contenedor restringe la accesibilidad de sus miembros.

Varias declaraciones de una [clase parcial](#) o un [miembro parcial](#) deben tener la misma accesibilidad. Si una declaración de la clase o miembro parcial no incluye un modificador de acceso, las demás declaraciones no pueden declarar un modificador de acceso. El compilador genera un error si varias declaraciones para la clase o método parcial declaran distintas accesibilidades.

Accesibilidad de clases y estructuras

Las clases y estructuras que se declaran directamente en un espacio de nombres (que no están anidadas en otras clases o estructuras) pueden tener acceso `public`, `internal` o `file`. 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 estructura, incluidas las clases y las estructuras anidadas, tienen acceso `private` de forma predeterminada.

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](#).

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. Los modificadores de acceso se usan en los miembros `interface` para proporcionar una implementación común necesaria para todos los implementadores de una interfaz.

Un tipo `delegate` declarado directamente en un espacio de nombres tiene acceso `internal` de forma predeterminada.

Para más información sobre los modificadores de acceso, consulte la página [Niveles de accesibilidad](#).

Accesibilidad de miembros

Los miembros de `class` o `struct` (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. Sin embargo, un miembro `public` de una clase `internal` 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](#).

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.

```
C#  
  
// 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; }  
    }  
}
```

Los finalizadores no pueden tener modificadores de accesibilidad. Los miembros de un tipo `enum` siempre son `public` y no se les puede aplicar ningún modificador de acceso.

El modificador de acceso `file` solo se permite en declaraciones de tipo de nivel superior (no anidadas).

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 orden del modificador (regla de estilo IDE0036)
- El sistema de tipos de C#
- Interfaces
- Niveles de accesibilidad
- private
- public
- internal
- protected
- protected internal
- private protected
- sealed
- clase
- struct
- interface
- Tipos anónimos (Guía de programación de C#).

Campos (Guía de programación de C#)

Artículo • 01/06/2023

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 al propio tipo y se comparte entre todas las instancias de ese tipo. Solo puede acceder al campo estático mediante el nombre del tipo. Si obtiene acceso al campo estático mediante un nombre de instancia, obtendrá el error en tiempo de compilación [CS0176](#).

Por lo general, debe declarar una accesibilidad `private` o `protected` de los campos. Los datos que el tipo expone al código de cliente se deben proporcionar mediante [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*. Puede declarar campos como `public`, pero entonces no podrá impedir que el código que usa su tipo establezca ese campo en un valor no válido o cambie los datos de un objeto.

Los campos almacenan habitualmente los datos que deben ser accesibles para más de un método de tipo y que deben almacenarse durante más tiempo de lo que dura un único método. Por ejemplo, es posible que un tipo 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 o estructura mediante la especificación del nivel de acceso, seguido por el tipo y el nombre del campo. Por ejemplo:

```
C#  
  
public class CalendarEntry  
{  
  
    // private field (Located near wrapping "Date" property).  
    private DateTime _date;  
  
    // 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("Date");
        }
    }
}

// public field (Generally not recommended).
public string? Day;

// 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("dateString");
    }
}

public TimeSpan GetTimeSpan(string dateString)
{
    DateTime dt = Convert.ToDateTime(dateString);

    if (dt.Ticks < _date.Ticks)
    {
        return _date - dt;
    }
    else
    {
        throw new ArgumentOutOfRangeException("dateString");
    }
}
```

Para acceder a un campo en una instancia, agregue un punto después del nombre de instancia, seguido del nombre del campo, como en `instancename._fieldName`. Por ejemplo:

C#

```
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:

C#

```
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, sobrescribe cualquier valor dado durante la declaración del campo. Para obtener más información, vea [Utilizar constructores](#).

ⓘ Nota

Un inicializador de campo no puede hacer referencia a otros campos de instancia.

Los campos se pueden marcar como `public`, `private`, `protected`, `internal`, `protected internal` o `private protected`. Estos modificadores de acceso definen cómo los usuarios del tipo pueden acceder a los campos. Para obtener más información, consulte [Modificadores de acceso](#).

Opcionalmente, un campo se puede declarar como `static`. El campo estático está disponible para los autores de la llamada en cualquier momento, aunque no exista ninguna instancia del tipo. Para más información, vea [Clases estáticas y sus miembros](#).

Un campo se puede declarar 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 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).

Un campo se puede declarar como `required`. Un campo obligatorio lo debe inicializar el constructor o un [inicializador de objetos](#) cuando se crea un objeto. Agregue el atributo `System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` a cualquier declaración de constructor que inicialice todos los miembros necesarios.

El modificador `required` no se puede combinar con el modificador `readonly` en el mismo campo. Sin embargo, una [propiedad](#) solo puede ser `required` e `init`.

A partir de C# 12, los parámetros de [constructor principal](#) son una alternativa a la declaración de campos. Cuando el tipo tiene dependencias que se deben proporcionar en la inicialización, puede crear un constructor principal que proporcione esas dependencias. Estos parámetros se pueden capturar y usar en lugar de campos declarados en los tipos. En el caso de los tipos [record](#), los parámetros de constructor principal se muestran como propiedades públicas.

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#.

Consulte también

- [Guía de programación de C#](#)
- [El sistema de tipos de C#](#)
- [Utilizar constructores](#)
- [Herencia](#)
- [Modificadores de acceso](#)
- [Clases y miembros de clase abstractos y sellados](#)

Constantes (Guía de programación de C#)

Artículo • 13/03/2024

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#](#) se pueden declarar como `const`. Las constantes de tipo de referencia que no sean `String` solo se pueden inicializar con un `valor null`. 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:

```
C#  
  
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.

ⓘ Nota

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:

```
C#
```

```
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:

```
C#
```

```
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 internas](#) o [privadas protegidas](#). Estos modificadores de acceso definen cómo los usuarios de la clase pueden acceder a la constante. Para 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:

```
C#
```

```
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#.

Consulte también

- [Propiedades](#)
- [Tipos](#)
- [readonly](#)
- [Inmutabilidad en la primera parte de C#: tipos de inmutabilidad](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento para definir propiedades abstractas (Guía de programación de C#)

Artículo • 07/04/2023

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`.

Ejemplos

Este archivo declara la clase `Shape` que contiene la propiedad `Area` del tipo `double`.

```
C#  
  
// 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:

C#

```

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.

En el siguiente código se muestran tres subclases de `Shape` y cómo invalidan la propiedad `Area` para proporcionar su propia implementación.

C#

```

// 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;
        }
    }
}
```

```
    }  
}
```

En el siguiente código se muestra un programa de prueba que crea un número de objetos derivados de `Shape` e imprime sus áreas.

C#

```
// 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#](#)
- [El sistema de tipos de C#](#)
- [Clases y miembros de clase abstractos y sellados](#)
- [Propiedades](#)

Definición de constantes en C#

Artículo • 07/04/2023

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.

ⓘ Nota

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

C#

```
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
        Console.WriteLine(secsFromSun);
    }
}
```

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.

Consulte también

- [El sistema de tipos de C#](#)

Propiedades (Guía de programación de C#)

Artículo • 26/10/2024

Una *propiedad* es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo de datos. Las propiedades aparecen como miembros de datos públicos, pero se implementan como métodos especiales llamados *descriptores de acceso*. Esta característica permite a las personas que llaman acceder fácilmente a los datos y, al mismo tiempo, contribuye a promover la seguridad y la flexibilidad de los datos. La sintaxis para propiedades es una extensión natural de los campos. Un campo define una ubicación de almacenamiento:

```
C#  
  
public class Person  
{  
    public string? FirstName;  
  
    // Omitted for brevity.  
}
```

Propiedades implementadas automáticamente

Una definición de propiedad contiene las declaraciones para un descriptor de acceso `get` y `set` que recupera y asigna el valor de esa propiedad:

```
C#  
  
public class Person  
{  
    public string? FirstName { get; set; }  
  
    // Omitted for brevity.  
}
```

En el ejemplo anterior se muestra una *propiedad* implementada automáticamente. El compilador genera un campo de respaldo oculto para la propiedad. El compilador también implementa el cuerpo de los descriptores de acceso `get` y `set`. Los atributos se aplican a la propiedad implementada automáticamente. Para aplicar el atributo al campo de respaldo generado por el compilador, puede especificar la etiqueta `field:` en el atributo.

Para inicializar una propiedad con un valor distinto del predeterminado, puede establecer un valor después del corchete 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 en el código siguiente:

```
C#  
  
public class Person  
{  
    public string FirstName { get; set; } = string.Empty;  
  
    // Omitted for brevity.  
}
```

Control de acceso

En los ejemplos anteriores se muestran propiedades de lectura y escritura. También 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`:

```
C#  
  
public class Person  
{  
    public string? FirstName { get; private set; }  
  
    // Omitted for brevity.  
}
```

La propiedad `FirstName` se puede leer 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. Un modificador de acceso en un descriptor de acceso individual debe ser más restrictivo que el acceso de la propiedad. El código 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`.

Hay dos modificadores de acceso especiales para los descriptores de acceso `set`:

- Un descriptor de acceso `set` puede tener `init` como modificador de acceso. Ese descriptor de acceso `set` solo se puede llamar desde un inicializador de objeto o los constructores del tipo. Es más restrictivo que `private` en el descriptor de acceso `set`.
- Una propiedad implementada automáticamente puede declarar un `get` descriptor de acceso sin un `set` descriptor de acceso. En ese caso, el compilador permite llamar al descriptor de acceso `set` solo desde los constructores del tipo. Es más restrictivo que el descriptor de acceso `init` en el descriptor de acceso `set`.

Modifique la clase `Person` de la manera siguiente:

```
C#
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

En el ejemplo anterior se requieren llamadas para usar el constructor que incluye el parámetro `FirstName`. Los autores de llamadas no pueden usar [inicializadores de objetos](#) para asignar un valor a la propiedad. Para admitir inicializadores, puede convertir el descriptor de acceso `set` en un descriptor de acceso `init`, como se muestra en el código siguiente:

```
C#
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

Estos modificadores se suelen usar con el modificador `required` para forzar la inicialización adecuada.

Propiedades necesarias

En el ejemplo anterior se permite a un autor de la llamada crear un `Person` mediante el constructor predeterminado, sin establecer la propiedad `FirstName`. La propiedad cambió el tipo a una cadena que acepta *valores NULL*. A partir de C# 11, puede *requerir* que los autores de las llamadas establezcan una propiedad:

```
C#  
  
public class Person  
{  
    public Person() { }  
  
    [SetsRequiredMembers]  
    public Person(string firstName) => FirstName = firstName;  
  
    public required string FirstName { get; init; }  
  
    // Omitted for brevity.  
}
```

El código anterior realiza dos cambios en la clase `Person`. En primer lugar, la declaración de la propiedad `FirstName` incluye el modificador `required`. Esto significa que cualquier código que cree un nuevo `Person` debe establecer esta propiedad mediante un [inicializador de objeto](#). En segundo lugar, el constructor que toma un parámetro `firstName` tiene el atributo

`System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute`. Este atributo informa al compilador de que este constructor establece *todos* los miembros de `required`. Los autores de llamadas que usan este constructor no son necesarios para establecer propiedades `required` con un inicializador de objeto.

ⓘ Importante

No confunda `required` con *que no acepta valores NULL*. Es válido establecer una propiedad `required` en `null` o `default`. Si el tipo no acepta valores NULL, como `string` en estos ejemplos, el compilador emite una advertencia.

```
C#  
  
var aPerson = new Person("John");  
aPerson = new Person{ FirstName = "John"};  
// Error CS9035: Required member `Person.FirstName` must be set:  
//aPerson2 = new Person();
```

Definiciones de cuerpos de expresión

Los descriptores de acceso de propiedad suelen constar de instrucciones de una sola línea. Los descriptores de acceso 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 token `=>` seguido de la expresión que se va a asignar a la propiedad o a recuperar de ella.

Las propiedades de solo lectura pueden implementar el descriptor de acceso `get` como miembro con forma de expresión. En el ejemplo siguiente se implementa la propiedad de solo lectura `Name` como miembro con forma de expresión.

```
C#  
  
public class Person  
{  
    public Person() {}  
  
    [SetsRequiredMembers]  
    public Person(string firstName, string lastName)  
    {  
        FirstName = firstName;  
        LastName = lastName;  
    }  
  
    public required string FirstName { get; init; }  
    public required string LastName { get; init; }  
  
    public string Name => $"{FirstName} {LastName}";  
  
    // Omitted for brevity.  
}
```

La propiedad `Name` es una propiedad calculada. No hay ningún campo de respaldo para `Name`. La propiedad la calcula cada vez.

Propiedades con campos de respaldo

Se puede combinar el concepto de una propiedad calculada con un campo privado y crear una *propiedad de evaluación en caché*. Por ejemplo, actualice la propiedad `FullName` para que el formato de cadena se produzca en el primer acceso:

```
C#  
  
public class Person  
{
```

```

public Person() { }

[SetsRequiredMembers]
public Person(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}

public required string FirstName { get; init; }
public required string LastName { get; init; }

private string? _fullName;
public string FullName
{
    get
    {
        if (_fullName is null)
            _fullName = $"{FirstName} {LastName}";
        return _fullName;
    }
}
}

```

Esta implementación funciona porque las propiedades `FirstName` y `LastName` son de solo lectura. Las personas pueden cambiar su nombre. La actualización de las propiedades `FirstName` y `LastName` para permitir a los descriptores de acceso `set` requiere que invalide cualquier valor almacenado en caché para `fullName`. Hay que modificar los descriptores de acceso `set` de la propiedad `FirstName` y `LastName` para que el campo `fullName` se calcule de nuevo:

C#

```

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 is null)
            _fullName = $"{FirstName} {LastName}";
        return _fullName;
    }
}
}

```

Esta versión final da como resultado la propiedad `FullName` solo cuando sea necesario. Se usa la versión calculada anteriormente si es válida. De lo contrario, el cálculo actualiza el valor almacenado en caché. 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`.

A partir de C# 13, puede crear propiedades `partial` en las clases `partial`. La declaración de implementación de una `partial` propiedad no puede ser una propiedad implementada automáticamente. Una propiedad implementada automáticamente usa la misma sintaxis que una declaración de propiedad parcial.

Propiedades

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.

- Las propiedades simples que no requieren código de descriptor de acceso personalizado se pueden implementar como definiciones de cuerpo de expresión o como [propiedades implementadas](#) automáticamente.
- 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`. Un descriptor de acceso de propiedad `init` se usa para asignar un nuevo valor solo durante la construcción del objeto. 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` o `init`.
- 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.

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#.

Consulte también

- [Indizadores](#)
- [palabra clave init](#)
- [get \(Palabra clave\)](#)
- [set \(palabra clave\)](#)

Utilizar propiedades (Guía de programación de C#)

Artículo • 20/08/2024

Las propiedades combinan aspectos de los campos y los métodos. Para el usuario de un objeto, una propiedad que parece un campo. Para acceder a la propiedad se 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` o `init`. 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` o `init` se ejecuta cuando se asigna un 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. Puede usar un descriptor de acceso `init` en lugar de un descriptor de acceso `set` para permitir que la propiedad se establezca como parte de la inicialización de objetos, pero de lo contrario, conviértalo en de solo lectura.

A diferencia de los campos, las propiedades no se clasifican como variables. Por lo tanto, no puede pasar una propiedad como un parámetro `ref` o `out`.

Las propiedades tienen muchos usos:

- Pueden validar los datos antes de permitir un cambio.
- Pueden exponer datos de forma transparente en una clase donde esos datos se recuperan de algún otro origen, como una base de datos.
- Pueden realizar una acción cuando se cambian los datos, 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:

```
C#  
  
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. La ubicación real de los datos de una propiedad se conoce a menudo como la "memoria auxiliar" de la propiedad. Es habitual que las propiedades usen campos privados como 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 sintaxis simplificada para declaraciones de propiedades simples. Para obtener más información, consulte [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. El compilador de C# y el compilador Just-In-Time (JIT) detectan patrones comunes para implementar el descriptor de acceso `get` y optimiza esos patrones. Por ejemplo, es probable que un descriptor de acceso `get` que devuelve un campo sin realizar ningún cálculo esté optimizado para una lectura de memoria de ese campo. Las propiedades implementadas automáticamente siguen este patrón y se benefician de estas optimizaciones. 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 ejemplo de un descriptor de acceso `get` que devuelve el valor de un campo privado `_name`:

C#

```
class Employee
{
    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:

```
C#  
  
var employee= new Employee();  
//...  
  
System.Console.WriteLine(employee.Name); // the get accessor is invoked here
```

El descriptor de acceso `get` debe ser un miembro con forma de expresión o finalizar en una instrucción `return` o `throw`, y el control no puede fluir fuera del cuerpo del descriptor de acceso.

⚠️ Advertencia

Cambiar el estado del objeto mediante el descriptor de acceso `get` es un estilo de programación incorrecto.

El descriptor de acceso `get` puede usarse para devolver el valor de campo o para calcularlo y devolverlo. Por ejemplo:

```
C#  
  
class Manager  
{  
    private string _name;  
    public string Name => _name != null ? _name : "NA";  
}
```

En el ejemplo anterior, si no asigna un valor a la propiedad `Name`, devuelve 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. El compilador y el compilador JIT también reconocen patrones comunes para un descriptor de acceso `set` o `init`. Esos patrones comunes están optimizados, escribiendo directamente la memoria para el campo de respaldo. En el siguiente ejemplo, se agrega un descriptor de acceso `set` a la propiedad `Name`:

```
C#
```

```
class Student
{
    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:

```
C#
var student = new Student();
student.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(student.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`.

El descriptor de acceso `init`

El código para crear un descriptor de acceso `init` es el mismo que para crear uno de tipo `set`, salvo que se usa la palabra clave `init` en lugar de `set`. La diferencia es que el descriptor de acceso `init` solo se puede usar en el constructor o mediante un [inicializador de objeto](#).

Observaciones

Las propiedades 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 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, consulte [Modificadores de acceso](#).

Una propiedad puede declararse como una propiedad estática mediante la palabra clave `static`. Las propiedades estáticas están disponibles para los autores de la llamada

en cualquier momento, aunque no exista ninguna instancia de la clase. Para más información, vea [Clases estáticas y sus miembros](#).

Una propiedad puede marcarse como una propiedad virtual mediante la palabra clave [virtual](#). Las propiedades virtuales permiten 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](#), lo que especifica que para las clases derivadas ya no es virtual. Por último, una propiedad puede declararse [abstracta](#). Las propiedades abstractas no definen 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 \(Guía de programación de C#\)](#).

ⓘ Nota

Es un error usar un modificador [virtual](#), [abstract](#) u [override](#) en un descriptor de acceso de una propiedad [static](#).

Ejemplos

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.

C#

```
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:  
}
```

Ejemplo de propiedad oculta

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:

C#

```
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  
{  
    public static void Test()  
    {  
        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:

```
C#  
  
public new string Name
```

- La conversión `(Employee)` se usa para tener acceso a la propiedad oculta de la clase base:

```
C#  
  
((Employee)m1).Name = "Mary";
```

Para obtener más información sobre cómo ocultar miembros, vea el [Modificador new](#).

Ejemplo de invalidación de propiedades

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.

```
C#  
  
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

- Propiedades
- Propiedades de interfaz
- Propiedades implementadas automáticamente
- Propiedades parciales

Propiedades de interfaces (Guía de programación de C#)

Artículo • 26/10/2024

Las propiedades se pueden declarar en una [interfaz](#). En el ejemplo siguiente se declara un descriptor de acceso de propiedad de interfaz:

C#

```
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 estructuras, declarar los descriptores de acceso sin un cuerpo no declara una [propiedad](#) implementada automáticamente. 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:

C#

```
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 es necesaria. Es decir, la siguiente declaración de propiedad:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

Implementa la propiedad `Name` en la interfaz `IEmployee`, mientras que la siguiente declaración:

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

Implementa la propiedad `Name` en la interfaz `ICitizen`.

C#

```
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;
}

```

C#

```

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);

```

Salida de ejemplo

Consola

```

Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous

```

Consulte también

- Propiedades

- Utilizar propiedades
- Comparación entre propiedades e indizadores
- Indizadores
- Interfaces

Restringir la accesibilidad del descriptor de acceso (Guía de programación de C#)

Artículo • 07/04/2023

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, restringe la accesibilidad del descriptor de acceso `set`, mientras que se mantiene el descriptor de acceso `get` accesible públicamente. Por ejemplo:

```
C#  
  
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.

ⓘ Nota

Los ejemplos de este artículo no usan **propiedades implementadas automáticamente**. Las *propiedades implementadas* automáticamente proporcionan una sintaxis concisa para declarar propiedades cuando no se requiere un campo de respaldo personalizado.

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:

C#

```
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:

C#

```
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 en la clase derivada. Genera un error al asignarlo o accede a la propiedad de clase base del mismo nombre, si es accesible.

C#

```
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

- [Propiedades](#)
- [Indizadores](#)
- [Modificadores de acceso](#)
- [Propiedades de solo inicialización](#)
- [Propiedades necesarias](#)

Procedimiento para declarar y usar propiedades de lectura y escritura (Guía de programación de C#)

Artículo • 07/04/2023

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. Las propiedades declaran 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

```
C#  
  
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;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }
}

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`. Los descriptores de acceso públicos permiten que cualquier objeto lea y escriba estas propiedades. En cambio, a veces esto es conveniente para excluir uno de los descriptores de acceso. Puede omitir el descriptor de acceso `set` para que la propiedad sea de solo lectura:

```
C#  
  
public string Name  
{  
    get  
    {  
        return _name;  
    }  
    private set  
    {  
        _name = value;  
    }  
}
```

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 declaran las propiedades, pueden usarse como campos de la clase. Las propiedades permiten una sintaxis natural cuando ambos obtienen y establecen el valor de una propiedad, como se muestra en las instrucciones siguientes:

```
C#  
  
person.Name = "Joe";  
person.Age = 99;
```

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:

```
C#
```

```
_name = value;
```

Tenga en cuenta la sintaxis pura para incrementar la propiedad `Age` en un objeto

`Person`:

C#

```
person.Age += 1;
```

Si los métodos `set` y `get` independientes se han usado para modelar las propiedades, el código equivalente puede tener este aspecto:

C#

```
person.SetAge(person.GetAge() + 1);
```

El método `ToString` se invalida en este ejemplo:

C#

```
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`.

Consulte también

- [Propiedades](#)
- [El sistema de tipos de C#](#)

Propiedades implementadas automáticamente

Artículo • 26/10/2024

Las propiedades implementadas automáticamente hacen que la declaración de propiedad sea más concisa cuando no se requiere ninguna otra lógica en los descriptores de acceso de 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`. Los descriptores de acceso también se pueden declarar como propiedades implementadas automáticamente.

Ejemplo

En el ejemplo siguiente se muestra una clase simple que tiene algunas propiedades implementadas automáticamente:

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public 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()
{
    // Initialize 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.

Puede inicializar propiedades implementadas automáticamente de forma similar a los campos:

C#

```
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 las clases pequeñas o estructuras que solo encapsulan un conjunto de valores (datos) y tienen poco o ningún comportamiento, debe usar una de las opciones siguientes para hacer que los objetos sean inmutables:

- Declare solo un descriptor de acceso `get` (inmutable en cualquier lugar excepto el constructor).
- Declare un descriptor de acceso `get` y un descriptor de acceso `init` (inmutable en cualquier lugar excepto durante la construcción del objeto).
- Declare el descriptor de acceso `set` como `private` (inmutable para los consumidores).

Para obtener más información, vea [Cómo implementar una clase ligera con propiedades implementadas automáticamente](#).

Consulte también

- Uso de propiedades implementadas automáticamente (regla de estilo IDE0032)
- Propiedades
- Modificadores

Cómo implementar una clase ligera con propiedades implementadas automáticamente

Artículo • 26/10/2024

En este ejemplo se muestra cómo crear una clase ligera inmutable que solo sirve para encapsular un conjunto de propiedades implementadas automáticamente. Use este tipo de construcción en lugar de un struct cuando deba utilizar una semántica de tipo de referencia.

Puede hacer que una propiedad sea inmutable de estas maneras:

- Declare solo el descriptor de acceso `get`, que hace que la propiedad sea inmutable en cualquier lugar excepto en el constructor del tipo.
- Declare un descriptor de acceso `init` en lugar de `set`, que hace que la propiedad se pueda establecer solo en el constructor o mediante un [inicializador de objeto](#).
- Declare el descriptor de acceso `set` como `private`. La propiedad solo se puede establecer dentro del tipo, pero es inmutable para los consumidores.

Puede agregar el modificador `required` a la declaración de propiedad para forzar a los llamadores a establecer la propiedad como parte de la inicialización de un nuevo objeto.

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.

C#

```
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 ejemplo siguiente se muestran dos maneras de implementar una clase inmutable que haya implementado automáticamente las propiedades. 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.

C#

```
// 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";
    }
}

/* 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 implementada automáticamente. No se puede acceder a los campos directamente desde el código fuente.

Consulte también

- [Propiedades](#)
- [struct](#)
- [Inicializadores de objeto y colección](#)

Métodos (Guía de programación de C#)

Artículo • 07/04/2023

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. En una aplicación que usa [instrucciones de nivel superior](#), el compilador genera el método `Main` y contiene todas las instrucciones de nivel superior.

ⓘ Nota

En este artículo se analizan los métodos denominados. Para obtener información sobre las funciones anónimas, consulte [Expresiones lambda](#).

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.

ⓘ Importante

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:

C#

```
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:

C#

```

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:

```
C#  
  
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 evaluates 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:

```
C#  
  
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`:

```
C#  
  
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 que aparece antes del nombre de método) no es `void`, el método puede

devolver el valor mediante la [instrucción 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 se puede devolver al autor de la llamada por valor o [por 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 una variable denominada `estDistance` mediante referencia al autor de la llamada.

C#

```
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:

C#

```
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:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
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:

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

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:

C#

```
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;
    }
}
```

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.

ⓘ Nota

Un método asincrónico vuelve al autor de la 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 normalmente tiene un tipo de valor devuelto de `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` 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. Un método asincrónico puede tener [cualquier tipo de valor devuelto similar a una tarea](#).

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 `Main` es un ejemplo de método asincrónico con un tipo de valor devuelto `Task`. Va al método `DoSomethingAsync`, y como se expresa con una sola línea, puede omitir las palabras clave `async` y `await`. 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();`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static 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();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example 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, consulte los artículos [Programación asincrónica con `async` y `await`](#) y [Tipos de valor devueltos asincrónicos](#).

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 `=>`:

C#

```
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 asíncrono, 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 utiliza 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.

Llame a un iterador a partir del código de cliente mediante una instrucción `foreach`.

El tipo de valor devuelto de un enumerador puede ser `IEnumerable`, `IEnumerable<T>`, `IAsyncEnumerable<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#](#)
- [El sistema de tipos de C#](#)
- [Modificadores de acceso](#)
- [Clases estáticas y sus miembros](#)
- [Herencia](#)
- [Clases y miembros de clase abstractos y sellados](#)
- [params](#)

- `out`
- `ref`
- Parámetros de métodos

Funciones locales (Guía de programación de C#)

Artículo • 26/04/2023

Las *funciones locales* son métodos 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.

ⓘ Nota

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:

C#

```
<modifiers> <return-type> <method-name> <parameter-list>
```

ⓘ Nota

`<parameter-list>` no debe contener los parámetros denominados con una **palabra clave contextual** `value`. El compilador crea la variable temporal "value", que contiene las variables externas a las que se hace referencia, lo que posteriormente provoca ambigüedad y también puede provocar un comportamiento inesperado.

Puede usar los siguientes modificadores con una función local:

- `async`
- `unsafe`
- `static` Una función local estática no puede capturar variables locales o el estado de la instancia.
- `extern` Una función local externa debe ser `static`.

Todas las variables locales que se definen en el miembro contenedor, incluidos sus parámetros de método, son accesibles en la función local no estática.

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".

En el ejemplo siguiente, se define una función local denominada `AppendPathSeparator` que es privada a un método denominado `GetText`:

C#

```
private static string GetText(string path, string filename)
{
    var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
    var text = reader.ReadToEnd();
    return text;

    string AppendPathSeparator(string filepath)
    {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
    }
}
```

A partir de C# 9.0, puede aplicar atributos a una función local, sus parámetros y parámetros de tipo, como se muestra en el ejemplo siguiente:

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

En el ejemplo anterior se usa un [atributo especial](#) para ayudar al compilador en el análisis estático en un contexto que acepta valores NULL.

Excepciones y funciones locales

Una de las características útiles de las funciones locales es que pueden permitir que las excepciones aparezcan inmediatamente. Para los métodos de iterador, 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 de 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.

C#

```
public class IteratorWithoutLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110);
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs) // line 11
        {
```

```

        Console.WriteLine($"{x} ");
    }

}

public static IEnumerable<int> OddSequence(int start, int end)
{
    if (start < 0 || start > 99)
        throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
    if (end > 100)
        throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
    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 output like this:
//
// Retrieved enumerator...
// Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
// at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
// at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11

```

Si coloca la lógica de iterador en una función local, se iniciarán excepciones de validación de argumentos al recuperar el enumerador, como se muestra en el ejemplo siguiente:

C#

```

public class IteratorWithLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110); // line 8
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs)
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)

```

```

{
    if (start < 0 || start > 99)
        throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
    if (end > 100)
        throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
    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 output like this:
//
// Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//   at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//   at IteratorWithLocalExample.Main() in IteratorWithLocal.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. Esta es la versión que usa una función local:

C#

```

public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}

```

Esta versión usa expresiones lambda:

C#

```
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);
}
```

Nomenclatura

Las funciones locales se nombran explícitamente como métodos. Las expresiones lambda son métodos anónimos y deben asignarse a variables de un tipo `delegate`, normalmente los tipos `Action` o `Func`. Cuando se declara una función local, el proceso es como escribir un método normal: se declaran un tipo de valor devuelto y una firma de función.

Signaturas de función y tipos de expresiones lambda

Las expresiones lambda se basan en el tipo de la variable `Action/Func` al que están asignadas para determinar los tipos de argumento y de valor devuelto. En las funciones locales, dado que la sintaxis se parece mucho a escribir un método normal, los tipos de argumento y el tipo de valor devuelto ya forman parte de la declaración de función.

A partir de C# 10, algunas expresiones lambda tienen un *tipo natural*, que permite al compilador deducir el tipo de valor devuelto y los tipos de parámetro de la expresión lambda.

Asignación definitiva

Las expresiones lambda son objetos que se declaran y se asignan en tiempo de ejecución. Para poder usar una expresión lambda, debe asignarse definitivamente: se debe declarar la variable `Action/Func` a la que se va a asignar y luego asignar a esta la expresión lambda. Observe que `LambdaFactorial` 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.

Las funciones locales se definen en tiempo de compilación. Dado que no están asignadas a variables, se puede hacer referencia a ellas desde cualquier ubicación del código **que esté en ámbito**; en el primer ejemplo `LocalFunctionFactorial`, se podría declarar la función local por encima o por debajo de la instrucción `return` y no desencadenar ningún error del compilador.

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.

Implementación como delegado

Las expresiones lambda se convierten en delegados en el momento en que se declaran. Las funciones locales son más flexibles, ya que se pueden escribir como un método tradicional o como un delegado. Las funciones locales solo se convierten en delegados 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.

Captura de variables

Las reglas de **asignación definitiva** también afectan a las variables capturadas por la función local o la expresión lambda. El compilador puede efectuar un análisis estático que permite a las funciones locales asignar definitivamente variables capturadas en el ámbito de inclusión. Considere este ejemplo:

```
C#  
  
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`.

Observe que cuando una función local captura variables en el ámbito de inclusión, la función local se implementa como un tipo delegado.

Asignaciones de montón

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 por 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:

C#

```
public async 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 await 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 dos asignaciones menos que la versión de la expresión lambda.

Si sabe que la función local no se va a convertir en delegado y ninguna de las variables capturadas por ella han sido capturadas por otras expresiones lambda o funciones locales que se han convertido en delegados, puede garantizar la no asignación de la función local al montón al declararla como función local `static`.

💡 Sugerencia

Habilite la regla de estilo de código de .NET [IDE0062](#) para asegurarse de que las funciones locales siempre estén marcadas como `static`.

⚠️ Nota

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`.

C#

```
public async 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 await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.";
    }
}
```

```
    Enjoy.";  
}  
}
```

Uso de la palabra clave `yield`

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.

C#

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)  
{  
    if (!input.Any())  
    {  
        throw new ArgumentException("There are no items to convert to  
lowercase.");  
    }  
  
    return LowercaseIterator();  
  
    IEnumerable<string> LowercaseIterator()  
    {  
        foreach (var output in input.Select(item => item.ToLower()))  
        {  
            yield return output;  
        }  
    }  
}
```

La instrucción `yield return` no está permitida en las expresiones lambda. Para más información, consulte el [error del compilador CS1621](#).

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.

Especificación del lenguaje C#

Para obtener más información, consulte la sección [Declaraciones de función local](#) de la [especificación del lenguaje C#](#).

Vea también

- [Uso de la función local en lugar de lambda \(regla de estilo IDE0039\)](#)
- [Métodos](#)

Variables locales con asignación implícita de tipos (Guía de programación de C#)

Artículo • 14/03/2023

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`:

C#

```
// 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](#).

```
C#
```

```
for (var x = 1; x < 10; x++)
```

- En una instrucción de inicialización [foreach](#).

```
C#
```

```
foreach (var item in list) {...}
```

- En una instrucción [using](#).

```
C#
```

```
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`.

```
C#
```

```
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:

```
C#
```

```
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:

```
C#
```

```
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.

```
C#
```

```
// 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 \(Language Integrated Query\)](#)
- [Instrucciones de iteración](#)
- [using \(Instrucción\)](#)

Procedimiento para usar matrices y variables locales con tipo implícito en expresiones de consulta (Guía de programación de C#)

Artículo • 07/04/2023

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](#).

Ejemplos

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](#).

C#

```
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);
```

```
    }  
}
```

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.

C#

```
// 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 \(Language Integrated Query\)](#)
- [LINQ en C#](#)

Métodos de extensión (Guía de programación de C#)

Artículo • 15/03/2024

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](#).

C#

```
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 sigue [este](#) modificador. 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:

C#

```
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`:

C#

```
using ExtensionMethods;
```

También se le puede llamar desde una aplicación con esta sintaxis:

C#

```
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.

Tanto la clase `MyExtensions` como el método `WordCount` son `static`, y se puede acceder a ellos como a todos los demás miembros `static`. El método `WordCount` se puede invocar como otros métodos `static`, como se muestra a continuación:

C#

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

El código de C# anterior:

- Declara y asigna un nuevo objeto `string` denominado `s` con un valor de `"Hello Extension Methods"`.
- Llama a `MyExtensions.WordCount` dado el argumento `s`.

Para más información, consulte [Implementación e invocación de 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:

C#

```
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 firma, 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 se encuentra una coincidencia, buscará cualquier método de extensión definido para el tipo y se enlazará al primer método de extensión que encuentre.

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 firma coinciden exactamente con los métodos ya implementados por las clases.

Si el compilador no encuentra un método de instancia con una firma coincidente, se enlazará a un método de extensión coincidente, si existe.

C#

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    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;

    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.

C#

```
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 del objeto `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. Puede agregar el modificador `ref` al primer argumento, lo que lo convierte en un método de extensión `ref`. La palabra clave `ref` puede aparecer antes o después de la palabra clave `this` sin ninguna diferencia semántica. Agregar el modificador `ref` indica que el primer argumento se pasa por referencia. Esto le permite escribir métodos de extensión que cambian el estado de la estructura que se extiende (tenga en cuenta que los miembros privados no son accesibles). Solo se permiten los tipos de valor o los tipos genéricos restringidos a la estructura (consulte [struct la restricción](#) para obtener más información) como primer parámetro de un método de extensión `ref`. En el ejemplo siguiente se muestra cómo usar un método de extensión `ref` para modificar directamente un tipo integrado sin necesidad de reasignar el resultado o pasarlo a través de una función con la palabra clave `ref`:

```
C#  
  
public static class IntExtensions  
{  
    public static void Increment(this int number)  
        => number++;  
  
    // Take note of the extra ref keyword here  
    public static void RefIncrement(this ref int number)  
        => number++;  
}  
  
public static class IntProgram  
{  
    public static void Test()  
    {
```

```

        int x = 1;

        // Takes x by value leading to the extension method
        // Increment modifying its own copy, leaving x unchanged
        x.Increment();
        Console.WriteLine($"x is now {x}"); // x is now 1

        // Takes x by reference leading to the extension method
        // RefIncrement changing the value of x directly
        x.RefIncrement();
        Console.WriteLine($"x is now {x}"); // x is now 2
    }
}

```

En este ejemplo siguiente se muestran los métodos de extensión `ref` para los tipos de estructura definidos por el usuario:

C#

```

public struct Account
{
    public uint id;
    public float balance;

    private int secret;
}

public static class AccountExtensions
{
    // ref keyword can also appear before the this keyword
    public static void Deposit(ref this Account account, float amount)
    {
        account.balance += amount;

        // The following line results in an error as an extension
        // method is not allowed to access private members
        // account.secret = 1; // CS0122
    }
}

public static class AccountProgram
{
    public static void Test()
    {
        Account account = new()
        {
            id = 1,
            balance = 100f
        };

        Console.WriteLine($"I have ${account.balance}"); // I have $100
        account.Deposit(50f);
    }
}

```

```
        Console.WriteLine($"I have ${account.balance}"); // I have $150
    }
}
```

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:

- No se llama 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](#).

Consulte también

- [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)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento para implementar e invocar un método de extensión personalizado (Guía de programación de C#)

Artículo • 04/10/2024

En este artículo se muestra cómo implementar sus propios métodos de extensión para cualquier tipo de .NET. El código cliente puede usar sus métodos de extensión. Los proyectos cliente deben hacer referencia al ensamblaje que los contiene. Los proyectos cliente deben añadir 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 no puede estar anidada dentro de otro tipo y debe ser visible para el código 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 métodos de instancia en el tipo.

ⓘ Nota

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`.

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`.

```
C#  
  
using CustomExtensions;  
  
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);  
  
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;  
        }  
    }  
}
```

La resolución de sobrecargas prefiere los métodos de instancia o estáticos definidos por el propio tipo a los métodos de extensión. Los métodos de extensión no pueden tener acceso a los datos privados de la clase extendida.

Consulte también

- [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#)

Artículo • 13/03/2024

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.

C#

```
using System;

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.

Consulte también

- [Métodos de extensión](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Argumentosopcionales y con nombre (Guía de programación de C#)

Artículo • 21/03/2024

Los *argumentos con nombre* permiten especificar un argumento para un parámetro asociando el argumento a su nombre y no a su posición 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 permiten proporcionar argumentos para los parámetros seleccionados. 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 hacer coincidir el orden de los argumentos con el de los parámetros en las listas de parámetros de los métodos llamados. El argumento de cada parámetro se puede especificar por el nombre del parámetro. Por ejemplo, se puede llamar 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.

C#

```
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.

C#

```
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,

```
C#
```

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- 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.

```
C#
```

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Los argumentos posicionales que siguen a los argumentos con nombre desordenados no son válidos.

```
C#
```

```
// 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.

```
C#
```

```
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");
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // 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}");
}
}

```

Argumentosopcionales

La definición de un método, constructor, indexador o delegado puede especificar que sus parámetros son necesarios u 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 parámetrosopcionales, debe proporcionar argumentos para todos los parámetrosopcionales 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.

C#

```
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.

C#

```
//anExample.ExampleMethod(3, ,4);
```

Pero si conoce el nombre del tercer parámetro, puede usar un argumento con nombre para realizar la tarea.

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

En IntelliSense los corchetes se usan para indicar parámetrosopcionales, como se muestra en la ilustración siguiente:

```
anExample.ExampleMethod(
    void ExampleClass.ExampleMethod(int required,
        [string optionalstr = "default string"],
        [int optionalint = 10])
```

ⓘ Nota

También puede declarar parámetrosopcionales con la clase [OptionalAttribute](#) de .NET. Los parámetros [OptionalAttribute](#) no requieren un valorpredeterminado. Sin embargo, si se desea un valorpredeterminado, eche un vistazo a la clase [DefaultParameterValueAttribute](#).

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.

C#

```
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
            // 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(
            $"({_name}): {required}, {optionalstr}, and {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.
}

```

En el código anterior se muestran varios ejemplos en los que los parámetros opcionales no se aplican correctamente. En primer lugar, se muestra que se debe proporcionar un argumento para el primer parámetro, que es obligatorio.

Atributos de información del autor de la llamada

Los [atributos de información del autor de la llamada](#), como [CallerFilePathAttribute](#), [CallerLineNumberAttribute](#), [CallerMemberNameAttribute](#) y [CallerArgumentExpressionAttribute](#), se usan para obtener información sobre el autor de la llamada a un método. Estos atributos son especialmente útiles cuando se depura o cuando se necesita registrar información sobre las llamadas de método.

Estos atributos son parámetros opcionales con valores predeterminados que proporciona el compilador. El autor de la llamada no debe proporcionar explícitamente un valor para estos parámetros.

Interfaces COM

Los argumentos opcionales y con nombre, además de compatibilidad con objetos dinámicos, 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])
```

Pero la llamada a `AutoFormat` se puede simplificar considerablemente mediante argumentos opcionales y con nombre. 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.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();  
excelApp.Workbooks.Add();  
excelApp.Visible = true;  
  
var myFormat =  
  
Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting  
1;  
  
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# \(Guía de programación de C#\)](#).

Resolución de sobrecarga

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. Generalmente, la resolución de sobrecarga prefiere aquellos candidatos que tengan 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#.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Constructores (guía de programación de C#)

Artículo • 06/03/2024

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 [Constructores de instancias](#) y [Uso de constructores](#).

Hay varias acciones que forman parte de la inicialización de una nueva instancia. Estas acciones tienen lugar en el orden siguiente:

1. *Campos de instancia se establecen en 0.* Normalmente, el tiempo de ejecución lo hace.
2. *Los inicializadores de campo se ejecutan.* Inicializadores de campo en la ejecución del tipo más derivado.
3. *Inicializadores de campo de tipo base ejecutan.* Inicializadores de campo a partir de la base directa a través de cada tipo base para [System.Object](#).
4. *Los constructores de instancia base se ejecutan.* Cualquier constructor de instancia, empezando por [Object.Object](#) a través de cada clase base a la clase base directa.
5. *El constructor de instancia se ejecuta.* El constructor estático para el tipo se ejecuta.
6. *Los inicializadores de objeto ejecutan.* Si la expresión incluye cualquier inicializador de objeto, se ejecutan después de que se ejecute el constructor de instancia. Los inicializadores de objeto se ejecutan en el orden textual.

Las acciones anteriores se realizan cuando se inicializa una nueva instancia. Si se establece una nueva instancia de un [struct](#) en su [default](#) valor, todos los campos de instancia se establecen en 0.

Si el [constructor estático](#) no se ha ejecutado, el constructor estático se ejecuta antes de que se realice ninguna de las acciones del constructor de instancia.

Sintaxis del constructor

Un constructor es un método cuyo nombre es igual que el nombre de su tipo. Su firma del método incluye solo un [modificador de acceso](#) opcional, 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](#).

C#

```
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`.

C#

```
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.

C#

```
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.
}
```

También puede definir un constructor estático con una definición de cuerpo de expresión, como se muestra en el ejemplo siguiente.

C#

```
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

- [Uso de constructores](#)
- [Constructores de instancias](#)
- [Constructores privados](#)
- [Constructores estáticos](#)
- [Escritura de un constructor de copia](#)

Consulte también

- [Guía de programación de C#](#)
- [El sistema de tipos de C#](#)
- [Finalizadores](#)
- [static](#)
- [Why Do Initializers Run In The Opposite Order As Constructors? Part One \(¿Por qué los inicializadores se ejecutan en orden contrario a los constructores? Parte uno\)](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Utilizar constructores (Guía de programación de C#)

Artículo • 01/06/2023

Cuando se crea una instancia de 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.

C#

```
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`. C# 12 presenta el concepto de *constructores principales*. Un constructor principal especifica parámetros que se deben proporcionar para inicializar un objeto nuevo. 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 más información, vea [Clases estáticas y sus miembros](#).

Puede impedir que se cree una instancia de una clase convirtiendo el constructor en privado, de la manera siguiente:

```
C#  
  
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 los tipos [struct](#) se asemejan a los constructores de clase. Cuando se crea una instancia de un tipo de struct con `new`, se invoca un constructor. Cuando `struct` se establece en su valor `default`, el runtime inicializa toda la memoria del struct en 0. Antes de la versión C# 10, `structs` no podía contener un constructor sin parámetros explícito, ya que el compilador proporcionaba uno automáticamente. Para más información, consulte la sección [Inicialización de estructuras y valores predeterminados](#) del artículo [Tipos de estructuras](#).

En el siguiente código se usa el constructor sin parámetros de [Int32](#), por lo que se tiene la certeza de que el entero se inicializa:

```
C#  
  
int i = new int();  
Console.WriteLine(i);
```

Sin embargo, el siguiente código genera un error del compilador porque no usa `new` y porque intenta usar un objeto que no se ha inicializado:

```
C#  
  
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:

C#

```
int a = 44; // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Tanto las clases como los structs pueden definir constructores que toman parámetros, [constructores principales](#) incluidos. Los constructores que toman parámetros deben llamarse mediante una instrucción `new` o `base`. Las clases y los structs también pueden definir varios constructores y no es necesario ninguno de ellos para definir un constructor sin parámetros. Por ejemplo:

C#

```
public class Employee
{
    public int Salary;

    public Employee() { }

    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:

C#

```
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:

C#

```
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. En la práctica, las siguientes declaraciones de constructor son iguales:

C#

```
public Manager(int initialData)  
{  
    //Add further instructions here.  
}
```

C#

```
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`:

C#

```
public Employee(int weeklySalary, int numberOfWeeks)  
    : this(weeklySalary * numberOfWeeks)  
{  
}
```

El uso de la palabra clave `this` en el ejemplo anterior llama a este constructor:

C#

```
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, consulte [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 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#.

Consulte también

- [Guía de programación de C#](#)
- [El sistema de tipos de C#](#)
- [Constructores](#)
- [Finalizadores](#)

Constructores de instancias (guía de programación de C#)

Artículo • 04/06/2024

Un constructor de instancia se declara para especificar el código que se ejecuta al crear una instancia de un tipo con la [expresión new](#). Para inicializar una clase [estática](#), o variables estáticas en una clase no estática, puede definir un [constructor estático](#).

Como se muestra en el ejemplo siguiente, puede declarar varios constructores de instancia en un tipo:

```
C#  
  
class Coords  
{  
    public Coords()  
        : this(0, 0)  
    { }  
  
    public Coords(int x, int y)  
    {  
        X = x;  
        Y = y;  
    }  
  
    public int X { get; set; }  
    public int Y { get; set; }  
  
    public override string ToString() => $"({X},{Y})";  
}  
  
class Example  
{  
    static void Main()  
    {  
        var p1 = new Coords();  
        Console.WriteLine($"Coords #1 at {p1}");  
        // Output: Coords #1 at (0,0)  
  
        var p2 = new Coords(5, 3);  
        Console.WriteLine($"Coords #2 at {p2}");  
        // Output: Coords #2 at (5,3)  
    }  
}
```

En el ejemplo anterior, el primer constructor, sin parámetros, llama al segundo constructor con los dos argumentos igual a `0`. Para hacerlo, use la palabra clave `this`.

Al declarar un constructor de instancia en una clase derivada, puede llamar a un constructor de una clase base. Para ello, use la palabra clave `base`, como se muestra en el ejemplo siguiente:

C#

```
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() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
    }
}
```

```
// Output: Area of the cylinder = 86.39
    }
}
```

Constructores sin parámetros

Si una *clase* no tiene constructores de instancia explícitos, C# proporciona un constructor sin parámetros que puede usar para crear instancias de una instancia de esa clase, como se muestra en el ejemplo siguiente:

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output: Name: unknown, Age: 0
    }
}
```

Ese constructor inicializa los campos de instancia y las propiedades según los inicializadores correspondientes. Si un campo o una propiedad no tiene ningún inicializador, su valor se establece en el [valor predeterminado](#) del tipo de la propiedad o del campo. Si declara al menos un constructor de instancia en una clase, C# no proporciona un constructor sin parámetros.

Una estructura *tipo* siempre proporciona un constructor sin parámetros. El constructor sin parámetros es un constructor sin parámetros implícito que genera el valor predeterminado de un tipo o un constructor sin parámetros declarado explícitamente. Para más información, consulte la sección [Inicialización de estructuras y valores predeterminados](#) del artículo [Tipos de estructuras](#).

Constructores principales

A partir de C# 12, puede declarar un *constructor principal* en clases y structs. Para ello, hay que colocar los parámetros entre paréntesis después del nombre de tipo:

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

Los parámetros de un constructor principal están dentro del ámbito de todo el cuerpo del tipo declarativo. Pueden inicializar propiedades o campos, usarse como variables en métodos o funciones locales y pasarse a un constructor base.

Un constructor principal indica que estos parámetros son necesarios en cualquier instancia del tipo. Los constructores escritos explícitamente deben usar la sintaxis `this(...)` del inicializador para invocar al constructor principal. Así, se garantiza que todos los constructores asignan indiscutiblemente parámetros del constructor principal. En cualquier tipo `class` (incluidos los tipos `record class`), el constructor sin parámetros implícito no se emite cuando hay un constructor principal. En cualquier tipo `struct` (incluidos los tipos `record struct`), siempre se emite el constructor sin parámetros implícito, y siempre se inicializan todos los campos (incluidos los parámetros del constructor principal) en el patrón de 0 bits. Si se escribe un constructor sin parámetros explícito, este debe invocar al constructor principal. En ese caso, se puede especificar un valor diferente en los parámetros del constructor principal. En el siguiente código se muestran ejemplos de constructores principales.

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
    NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

Puede agregar atributos al método de constructor principal sintetizado especificando el destino `method:` en el atributo :

C#

```
[method: MyAttribute]  
public class TaggedWidget(string name)  
{  
    // details elided  
}
```

Si no especifica el destino `method`, el atributo se coloca en la clase en lugar del método.

En los tipos `class` y `struct`, los parámetros del constructor principal están disponibles en cualquier parte del cuerpo del tipo. El parámetro se puede implementar como un campo privado capturado. Si las únicas referencias a un parámetro son inicializadores y llamadas de constructor, ese parámetro no se captura en un campo privado. Los usos en otros miembros del tipo hacen que el compilador capture el parámetro en un campo privado.

Si el tipo incluye el modificador `record`, el compilador sintetiza en su lugar una propiedad pública con el mismo nombre que el parámetro de constructor principal. En los tipos `record class`, si un parámetro de constructor principal usa el mismo nombre que un constructor principal base, esa propiedad es una propiedad pública del tipo base `record class`. No se duplica en el tipo derivado `record class`. Estas propiedades no se generan con los tipos que no son `record`.

Consulte también

- [Clases, estructuras y registros](#)
- [Constructores](#)
- [Finalizadores](#)
- [base](#)
- [this](#)
- [Especificaciones de características de constructores principales](#)



Colaborar con nosotros en
[GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

[Abrir incidencia con la documentación](#)

nuestra guía para
colaboradores.

☒ Proporcionar comentarios sobre
el producto

Constructores privados (Guía de programación de C#)

Artículo • 13/03/2024

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:

```
C#  
  
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.

```
C#  
  
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:

C#

```
// Counter aCounter = new Counter(); // Error
```

Consulte también

- [El sistema de tipos de C#](#)
- [Constructores](#)
- [Finalizadores](#)
- [private](#)
- [public](#)



[Colaborar con nosotros en GitHub](#)

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

 Abrir incidencia con la documentación

 Proporcionar comentarios sobre el producto

Constructores estáticos (Guía de programación de C#)

Artículo • 05/08/2024

Un constructor estático se usa para inicializar cualquier dato [estático](#) o realizar una acción determinada que solo debe realizarse una vez. Se llama automáticamente antes de crear la primera instancia o de hacer referencia a cualquier miembro estático. Se llama a un constructor estático como máximo una vez.

C#

```
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;
    }
}
```

Hay varias acciones que forman parte de la inicialización estática. Estas acciones tienen lugar en el orden siguiente:

1. *campos estáticos se establecen en 0*. Normalmente, esta inicialización se realiza en tiempo de ejecución.
2. *Inicializadores de campo estático se ejecutan*. Inicializadores de campo estáticos en la ejecución de tipo más derivado.
3. *Inicializadores de campo estático de tipo base se ejecutan*. Inicializadores de campo estáticos a partir de la base directa a través de cada tipo base para [System.Object](#).
4. *Cualquier ejecución de un constructor estático*. Cualquier constructor estático, desde la clase base final de [Object](#) pasando por cada clase base hasta la ejecución de tipos. No se especifica el orden de ejecución del constructor estático. Sin embargo, todos los constructores estáticos de la jerarquía se ejecutan antes de que se creen instancias.

ⓘ Importante

Hay una excepción importante a la regla de que un constructor estático se ejecuta antes de que se cree cualquier instancia. Si un inicializador de campo estático crea

una instancia del tipo, ese inicializador se ejecuta (incluida cualquier llamada a un constructor de instancia) antes de que se ejecute el constructor estático. Esto es más común en el *patrón singleton*, como se muestra en el ejemplo siguiente:

```
C#  
  
public class Singleton  
{  
    // Static field initializer calls instance constructor.  
    private static Singleton instance = new Singleton();  
  
    private Singleton()  
    {  
        Console.WriteLine("Executes before static constructor.");  
    }  
  
    static Singleton()  
    {  
        Console.WriteLine("Executes after instance constructor.");  
    }  
  
    public static Singleton Instance => instance;  
}
```

Un [inicializador de módulo](#) puede ser una alternativa a un constructor estático. Para obtener más información, consulte la [especificación de para inicializadores de módulo](#).

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 destinado a que lo llame Common Language Runtime (CLR). Se invoca automáticamente.
- El usuario no puede controlar cuándo se ejecuta el constructor estático en el programa.
- A un constructor estático se le llama automáticamente. Inicializa la [clase](#) antes de crear la primera instancia o de hacer referencia a cualquier miembro estático declarado en esa clase (no sus clases base). Un constructor estático se ejecuta antes que un constructor de instancia. Si existen inicializadores de variables de campo estático en la clase del constructor estático, se ejecutan en el orden textual

en el que aparecen en la declaración de la clase. Los inicializadores se ejecutan inmediatamente antes 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 entorno de ejecución no lo invoca una segunda vez y el tipo permanece sin inicializar lo que dura el dominio de la aplicación. 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 no definidos de forma explícita en el código fuente, para solucionar problemas puede ser necesario exigir 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.
- El entorno de ejecución llama a un constructor estático no más de una vez en un dominio de aplicación única. Esa llamada se realiza en una región bloqueada en función del tipo específico de la clase. No se necesitan mecanismos de bloqueo adicionales en el cuerpo de un constructor estático. Para evitar el riesgo de interbloqueos, no bloquee el subproceso actual en constructores estáticos e inicializadores. Por ejemplo, no espere por tareas, subprocesos, identificadores de espera o eventos, no adquiera bloqueos y no ejecute operaciones en paralelo de bloqueo, como bucles paralelos, `Parallel.Invoke` y consultas de Parallel LINQ.

ⓘ Nota

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 de parámetros 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.

```
C#
```

```
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.ToString());
    }

    // 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.
        Console.WriteLine("Press any key to exit.");
        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#](#).

Consulte también

- [El sistema de tipos de C#](#)
- [Constructores](#)
- [Clases estáticas y sus miembros](#)
- [Finalizadores](#)
- [Instrucciones de diseño de constructores](#)
- [Advertencia de seguridad de - CA2121: los constructores estáticos deben ser privados](#)
- [Inicializadores de módulo](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento para escribir un constructor de copia (Guía de programación de C#)

Artículo • 13/03/2024

Los [registros](#) de C# proporcionan un constructor de copia para objetos, pero debe escribir uno para las clases personalmente.

ⓘ Importante

Escribir constructores de copia que funcionen para todos los tipos derivados de una jerarquía de clases puede ser difícil. Si la clase no está `sealed`, conviene considerar la posibilidad de crear una jerarquía de tipos `record class` para usar el constructor de copia sintetizado por el compilador.

Ejemplo

En el ejemplo siguiente, `Person` 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. La clase `Person` está `sealed`, por lo que no se pueden declarar tipos derivados que puedan introducir errores porque solo se copie la clase base.

C#

```
public sealed 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](#)
- [Registros](#)

- El sistema de tipos de C#
- Constructores
- Finalizadores

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Finalizadores (Guía de programación de C#)

Artículo • 14/03/2023

Los finalizadores (anteriormente conocidos como **destructores**) se usan para realizar cualquier limpieza final necesaria cuando el recolector de elementos no utilizados recopila una instancia de clase. En la mayoría de los casos, puede evitar escribir un finalizador mediante [System.Runtime.InteropServices.SafeHandle](#) o clases derivadas para encapsular cualquier identificador no administrado.

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`.

```
C#  
  
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.

```
C#  
  
public class Destroyer  
{  
    public override string ToString() => GetType().Name;  
  
    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer 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:

C#

```
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.

ⓘ Nota

Los finalizadores vacíos no deben usarse. Cuando una clase contiene un finalizador, se crea una entrada en la cola `Finalize`. El recolector de elementos no utilizados procesa esta cola. Cuando esto sucede, llama a cada finalizador. Los finalizadores innecesarios, como los vacíos, los que solo llaman al finalizador de clase base o los que solo llaman a métodos emitidos condicionalmente provocan una pérdida innecesaria de rendimiento.

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. 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.

ⓘ Nota

El hecho de que los finalizadores se ejecuten o no como parte de la finalización de aplicaciones es específico de cada **implementación de .NET**. Cuando finaliza una aplicación, .NET Framework hace todo lo posible para llamar a todos los

finalizadores en los objetos cuyos elementos no se han detectado durante la recolección de elementos no utilizados, a menos que se haya suprimido esa limpieza (por ejemplo, mediante una llamada al método de biblioteca `GC.SuppressFinalize`). En .NET 5 (incluido .NET Core) y versiones posteriores no se llama a los finalizadores como parte de la finalización de la aplicación. Para obtener más información, vea la incidencia de GitHub [dotnet/csharpstandard n.º 291](#).

Si necesita realizar la limpieza de forma confiable cuando existe una aplicación, registre un controlador para el evento `System.AppDomain.ProcessExit`. Ese controlador garantizaría la llamada a `IDisposable.Dispose()`, o a `IAsyncDisposable.DisposeAsync()`, para todos los objetos que requieren limpieza antes de que la aplicación se cierre. Dado que no se puede llamar directamente a *Finalizar* y no se puede garantizar que el recolector de elementos no utilizados llame a todos los finalizadores antes de salir, debe usar `Dispose` o `DisposeAsync` para asegurarse de que se liberan los recursos.

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](#)

- Implementación de un método DisposeAsync
- 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. La salida de este código depende de la implementación de .NET a la que se dirige la aplicación:

- .NET Framework: en la salida se muestra que se llama automáticamente a los finalizadores de las tres clases cuando finaliza la aplicación, en orden, del más derivado al menos.
- NET 5 (incluido .NET Core) o una versión posterior: no hay ninguna salida, porque esta implementación de .NET no llama a los finalizadores cuando finaliza la aplicación.

```
C#  
  
class First  
{  
    ~First()  
    {  
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");  
    }  
}  
  
class Second : First  
{  
    ~Second()  
    {  
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");  
    }  
}  
  
class Third : Second  
{  
    ~Third()  
    {  
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");  
    }  
}  
  
/*  
Test with code like the following:  
    Third t = new Third();  
    t = null;  
  
When objects are finalized, the output would be:
```

```
Third's finalizer is called.  
Second's finalizer is called.  
First's finalizer is called.  
*/
```

Especificación del lenguaje C#

Para más información, consulte la sección [Finalizadores](#) de la [especificación del lenguaje C#](#).

Consulte también

- [IDisposable](#)
- [Guía de programación de C#](#)
- [Constructores](#)
- [Recolección de elementos no utilizados](#)

Inicializadores de objeto y de colección (Guía de programación de C#)

Artículo • 17/05/2024

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 [Propiedades implementadas automáticamente](#).

C#

```
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;
    }
}
```

C#

```
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.

Los inicializadores de objeto pueden establecer índizadores, además de asignar campos y propiedades. Tenga en cuenta esta clase básica `Matrix`:

```
C#  
  
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:

```
C#  
  
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, los siguientes inicializadores siguientes son válidos si `IndexersExample` tiene los indizadores adecuados:

C#

```
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:

C#

```
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.

C#

```
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. Es posible que desee 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.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

Al ejecutarse esta consulta, la variable `productInfos` incluye una secuencia de objetos a la que se puede tener acceso en una instrucción `foreach`, como se muestra en este ejemplo:

C#

```
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`.

C#

```
select new {p.ProductName, Price = p.UnitPrice};
```

Inicializadores de objeto con el modificador required

La palabra clave `required` se usa para forzar a los autores de llamadas a establecer el valor de una propiedad o campo mediante un inicializador de objeto. No es necesario establecer las propiedades necesarias como parámetros de constructor. El compilador garantiza que todos los autores de llamadas inicialicen esos valores.

C#

```
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object
```

```
initializer or attribute constructor.  
// var pet = new Pet();
```

Es una práctica habitual garantizar que el objeto se inicialice correctamente, especialmente cuando tienen varios campos o propiedades por administrar y no desea incluirlos todos en el constructor.

Inicializadores de objeto con el descriptor de acceso `init`

Puede asegurarse de que nadie cambie el objeto diseñado al limitarlo mediante un descriptor de acceso `init`. Ayuda a restringir la configuración del valor de propiedad.

C#

```
public class Person  
{  
    public string FirstName { get; set; }  
    public string LastName { get; init; }  
}  
  
// The `LastName` property can be set only during initialization. It CAN'T  
be modified afterwards.  
// The `FirstName` property can be modified after initialization.  
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};  
  
// You can assign the FirstName property to a different value.  
pet.FirstName = "Jane";  
  
// Compiler error:  
// Error CS8852 Init - only property or indexer 'Person.LastName' can only  
be assigned in an object initializer,  
// or on 'this' or 'base' in an instance constructor or an  
'init' accessor.  
// pet.LastName = "Kowalski";
```

Las propiedades de solo inicialización necesarias admiten estructuras inmutables, al tiempo que permiten la sintaxis natural para los usuarios del tipo.

Inicializadores de objeto con propiedades con tipo de clase

Es fundamental tener en cuenta las implicaciones de las propiedades con tipo de clase al inicializar un objeto:

C#

```
public class HowToClassTypedInitializer
{
    public class EmbeddedClassTypeA
    {
        public int I { get; set; }
        public bool B { get; set; }
        public string S { get; set; }
        public EmbeddedClassTypeB ClassB { get; set; }

        public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

        public EmbeddedClassTypeA()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeA constructor.
Values are: {this}");
            I = 3;
            B = true;
            S = "abc";
            ClassB = new() { BB = true, BI = 43 };
            Console.WriteLine($"Exiting EmbeddedClassTypeA constructor.
Values are: {this}");
        }
    }

    public class EmbeddedClassTypeB
    {
        public int BI { get; set; }
        public bool BB { get; set; }
        public string BS { get; set; }

        public override string ToString() => $"{BI}|{BB}|{BS}";

        public EmbeddedClassTypeB()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeB constructor.
Values are: {this}");
            BI = 23;
            BB = false;
            BS = "BBBabc";
            Console.WriteLine($"Exiting EmbeddedClassTypeB constructor.
Values are: {this}");
        }
    }

    public static void Main()
    {
        var a = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = { BI = 100003 }
        };
    }
}
```

```

        Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

        var a2 = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = new() { BI = 100003 } //New instance
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA a2:
{a2}");
    }

    // Output:
    //Entering EmbeddedClassTypeA constructor Values are: 0|False|||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc
    //Exiting EmbeddedClassTypeA constructor Values are:
    3|True|abc|||43|True|BBBabc
    //After initializing EmbeddedClassTypeA:
    103|False|abc|||100003|True|BBBabc
    //Entering EmbeddedClassTypeA constructor Values are: 0|False|||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc
    //Exiting EmbeddedClassTypeA constructor Values are:
    3|True|abc|||43|True|BBBabc
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc
    //After initializing EmbeddedClassTypeA a2:
    103|False|abc|||100003|False|BBBabc
}

```

En el ejemplo siguiente se muestra cómo, para ClassB, el proceso de inicialización implica actualizar valores específicos mientras se conservan otros de la instancia original. El inicializador reutiliza la instancia actual: los valores de ClassB son: 100003 (nuevo valor que asignamos aquí), true (mantenido desde la inicialización de EmbeddedClassTypeA), BBBabc (sin modificar el valor predeterminado de EmbeddedClassTypeB).

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, 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:

C#

```
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. Los inicializadores de objeto individuales se escriben entre llaves y se separan por comas.

C#

```
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.

C#

```
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.

C#

```
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. Puede inicializar también 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:

C#

```
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.

Inicializadores de objeto con inicialización de propiedades de solo lectura de colección

Algunas clases pueden tener propiedades de colección donde la propiedad es de solo lectura, como la propiedad `Cats` de `CatOwner` en el caso siguiente:

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
```

No puede usar la sintaxis del inicializador de colección abordada hasta ahora, ya que no se puede asignar una nueva lista a la propiedad:

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

Sin embargo, se pueden agregar nuevas entradas a `Cats` mediante la sintaxis de inicialización omitiendo la creación de lista (`new List<Cat>`), como se muestra a continuación:

```
C#  
  
CatOwner owner = new CatOwner  
{  
    Cats =  
    {  
        new Cat{ Name = "Sylvester", Age=8 },  
        new Cat{ Name = "Whiskers", Age=2 },  
        new Cat{ Name = "Sasha", Age=14 }  
    }  
};
```

El conjunto de entradas que se van a agregar aparecen entre llaves. El código anterior es idéntico a la escritura:

```
C#  
  
CatOwner owner = new ();  
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });  
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });  
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Ejemplos

En el ejemplo siguiente se combinan los conceptos de inicializadores de objeto y colección.

```
C#  
  
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.
}

```

En el ejemplo siguiente se muestra un objeto que implementa `IEnumerable` y contiene un método `Add` con varios parámetros. Usa un inicializador de colección con varios elementos por elemento en la lista que corresponden a la signatura del método `Add`.

C#

```
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. A partir de C# 13, el parámetro `params` no está restringido a una matriz. Puede ser un tipo de colección o una interfaz.

C#

```
public class DictionaryExample  
{  
    class RudimentaryMultiValuedDictionary<TKey, TValue> :  
        IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull  
    {  
        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
*/
}
```

Consulte también

- [Uso de inicializadores de objetos \(regla de estilo IDE0017\)](#)
- [Uso de inicializadores de colección \(regla de estilo IDE0028\)](#)
- [LINQ en C#](#)
- [Tipos anónimos](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento para inicializar objetos usando un inicializador de objeto (Guía de programación de C#)

Artículo • 15/05/2024

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 sin parámetros 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`:

C#

```
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 parameterless
        // constructor is
        // invoked in processing this declaration, not the constructor that
        // has
        // two parameters.
        StudentName student2 =
        {

```

```

        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 parameterless 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
{
    // This constructor has no parameters. The parameterless 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:

C#

```

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"]);
}

```

```
    }  
}
```

En el ejemplo siguiente se muestra el orden de ejecución de inicializaciones de constructor y miembro mediante constructor con y sin parámetro:

C#

Consulte también

- [Inicializadores de objeto y colección](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimientos: inicialización de un diccionario con un inicializador de colección (guía de programación de C#)

Artículo • 04/07/2024

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.

ⓘ Nota

La diferencia principal entre estas dos formas de inicializar la colección es que, en el caso de tener claves duplicadas, por ejemplo:

```
C#  
  
{ 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211  
} },  
{ 111, new StudentName { FirstName="Dina", LastName="Salimzianova",  
ID=317 } },
```

el método `Add` iniciará `ArgumentException`: 'An item with the same key has already been added. Key: 111' , mientras que la segunda parte del ejemplo, el método de indexador de lectura y escritura público, sobrescribirá silenciosamente la entrada ya existente con la misma clave.

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`:

```
C#
```

```

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 más internas encierra el inicializador de objeto para el `StudentName` y las llaves más externas encierra el inicializador para el par clave-valor que se va a agregar al `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`.

Consulte también

- [Inicializadores de objeto y colección](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Tipos anidados (Guía de programación de C#)

Artículo • 07/04/2023

Un tipo definido en una [clase, estructura](#) o [interfaz](#) se denomina tipo anidado. Por ejemplo

```
C#  
  
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".

Tenga en cuenta también que la creación de un tipo anidado externamente visible infringe la regla de calidad del código [CA1034](#) "Los tipos anidados no deben ser visibles".

- Los tipos anidados de un [struct](#) pueden ser [public](#), [internal](#) o [private](#).

En el ejemplo siguiente se convierte la clase `Nested` en `public`:

```
C#  
  
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:

C#

```
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:

C#

```
Container.Nested nest = new Container.Nested();
```

Vea también

- [Guía de programación de C#](#)
- [El sistema de tipos de C#](#)
- [Modificadores de acceso](#)

- Constructores
- Regla CA1034

Clases y métodos parciales (Guía de programación de C#)

Artículo • 26/10/2024

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:

- El hecho de declarar una clase entre archivos independientes permite que varios programadores trabajen en ella al mismo tiempo.
- Puedes agregar código a la clase sin tener que volver a crear el archivo de origen que incluye el origen generado automáticamente. 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.
- Los [generadores de origen](#) pueden ofrecer función adicional en una clase.

Para dividir una definición de clase, use el modificador de palabra clave [partial](#), como se muestra aquí:

```
C#  
  
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.

ⓘ Nota

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.

```
C#  
  
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:

```
C#
```

```
[SerializableAttribute]  
partial class Moon { }
```

```
[ObsoleteAttribute]  
partial class Moon { }
```

Son equivalentes a las declaraciones siguientes:

C#

```
[SerializableAttribute]  
[ObsoleteAttribute]  
class Moon { }
```

A continuación se indican los elementos que se combinan de todas las definiciones de tipo parcial:

- Comentarios XML. Sin embargo, si ambas declaraciones de un miembro parcial incluyen comentarios, solo se incluyen los comentarios del miembro de implementación.
- interfaces
- atributos de parámetro de tipo genérico
- class (atributos)
- miembros

Por ejemplo, consideremos las siguientes declaraciones:

C#

```
partial class Earth : Planet, IRotate { }  
partial class Earth : IRevolve { }
```

Son equivalentes a las declaraciones siguientes:

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

Restricciones

Debes 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:

C#

```
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:

C#

```
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 de una definición de tipo parcial son opcionales, pero si están presentes en una definición de tipo parcial, deben especificarse en otra definición parcial del mismo tipo:
 - `pública`
 - `private`
 - `protected`
 - `internal`
 - `abstract`
 - `sealed`
 - `clase base`

- modificador `new` (elementos anidados)
- restricciones genéricas

Para obtener más información, vea [Restricciones de tipos de parámetros](#).

Ejemplos

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.

```
C#  
  
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
```

En el ejemplo siguiente se muestra que también se pueden desarrollar structs e interfaces parciales.

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Miembros parciales

Una clase o estructura parcial puede contener un miembro parcial. Una parte de la clase contiene la firma del miembro. Una implementación se puede definir en el mismo elemento o en otro.

No es necesaria una implementación para un método parcial cuando la firma cumple las reglas siguientes:

- La declaración no incluye ningún modificador de acceso. El método tiene acceso a `private` de manera predeterminada.
- El tipo devuelto es `void`.
- Ninguno de los parámetros tiene el modificador `out`.
- La declaración de método no puede incluir ninguno de los modificadores siguientes:
 - `virtual`
 - `override`
 - `sealed`
 - `new`
 - `extern`

Si no se proporciona la implementación, el método y todas las llamadas al método se quitan en tiempo de compilación.

Cualquier método que no cumpla todas estas restricciones, incluidas las propiedades y los indizadores, debe proporcionar una implementación. Esa implementación la podría proporcionar un *generador de código fuente*. Las propiedades parciales no se pueden implementar mediante propiedades implementadas automáticamente. El compilador no puede distinguir entre una propiedad implementada automáticamente y la declaración declaratoria de una propiedad parcial.

Los métodos parciales permiten que el implementador de una parte de una clase declare un miembro. El implementador de otra parte de la clase puede definir ese miembro. Hay dos escenarios en los que esta separación es útil: plantillas que generan código reutilizable y generadores de código fuente.

- **Código de plantilla:** la plantilla reserva un nombre de método y una firma para que el código generado pueda llamar al método. Estos métodos siguen las restricciones que permiten a un desarrollador decidir si implementar el método. 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 produce ningún error en tiempo de compilación o en tiempo de ejecución si se llama al método pero no se implementa.
- **Generadores de código fuente:** los generadores de código fuente proporcionan una implementación para los miembros. El desarrollador humano puede agregar la declaración de miembro (a menudo con atributos leídos por el generador de código fuente). El desarrollador puede escribir código que llame a estos miembros. El generador de código fuente se ejecuta durante la compilación y proporciona la implementación. En este escenario, no se suelen seguir las restricciones de los miembros parciales que pueden no implementarse.

```
C#  
  
// Definition in file1.cs  
partial void OnNameChanged();  
  
// Implementation in file2.cs  
partial void OnNameChanged()  
{  
    // method body  
}
```

- Las declaraciones de miembro parcial deben comenzar con la palabra clave contextual **partial**.

- Las signaturas de miembros parciales de los dos elementos del tipo parcial deben coincidir.
- Los miembros parciales pueden tener modificadores [static](#) y [unsafe](#).
- El miembro parcial puede ser genérico. Las restricciones deben ser las mismas en la declaración de método de la definición e 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.
- Puedes crear un [delegado](#) para un método parcial definido e implementado, pero no para un método parcial que no tiene una implementación.

Especificación del lenguaje C#

Para obtener más información, vea la sección [Tipos parciales y Métodos parciales](#) de la [Especificación del lenguaje C#](#). La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#. Las características adicionales para los métodos parciales se definen en la [especificación de características](#).

Consulte también

- [Clases](#)
- [Tipos de estructura](#)
- [Interfaces](#)
- [parcial \(tipo\)](#)

Procedimiento para devolver subconjuntos de propiedades de elementos en una consulta (Guía de programación de C#)

Artículo • 13/03/2024

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:

C#

```
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 coincide con la condición especificada.

C#

```
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:

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

Si lo intenta en el ejemplo anterior, también debe cambiar la instrucción

```
Console.WriteLine:
```

C#

```
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.

Consulte también

- [Tipos anónimos](#)
- [LINQ en C#](#)

 Colaborar con nosotros en
GitHub

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Implementación de interfaz explícita (Guía de programación de C#)

Artículo • 13/03/2024

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:

C#

```
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:

C#

```
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
```

Pero es posible que no quiera que se llame a la misma implementación para las dos interfaces. Para llamar a otra implementación en función de la interfaz en uso, puede implementar un miembro de interfaz de forma explícita. Una implementación de interfaz explícita es un miembro de clase al que solo se llama a través de la interfaz especificada. Asigne al miembro de clase el nombre de la interfaz y un punto como prefijo. Por ejemplo:

C#

```
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:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint(); // Calls IControl.Paint on SampleClass.
surface.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:

C#

```
interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Una implementación de interfaz explícita no tiene un modificador de acceso, ya que no es accesible como miembro del tipo en el que se define. En su lugar, solo es accesible cuando se llama mediante una instancia de la interfaz. Si especifica un modificador de acceso para una implementación de interfaz explícita, obtendrá el error del compilador [CS0106](#). Para obtener más información, vea [interface \(Referencia de C#\)](#).

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:

C#

```
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:

C#

```
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.

Consulte también

- [Programación orientada a objetos](#)
- [Interfaces](#)
- [Herencia](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento Implementar explícitamente miembros de interfaz (Guía de programación de C#)

Artículo • 13/03/2024

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

C#

```
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;
```

```

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`:

C#

```

//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:

C#

```

System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());

```

Consulte también

- [Programación orientada a objetos](#)
- [Interfaces](#)
- [Procedimiento para implementar miembros de dos interfaces de forma explícita](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento Implementar explícitamente miembros de dos interfaces (Guía de programación de C#)

Artículo • 13/03/2024

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

C#

```
// 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:

C#

```

// 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:

C#

```
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());
}
```

Consulte también

- Programación orientada a objetos
- Interfaces
- Procedimiento para implementar miembros de interfaz de forma explícita



Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

Abrir incidencia con la documentación

Proporcionar comentarios sobre el producto

Delegados (Guía de programación de C#)

Artículo • 11/04/2024

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:

C#

```
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. Esta flexibilidad significa que puede cambiar las llamadas de método mediante programación, o bien agregar código nuevo a las clases existentes.

ⓘ Nota

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. Puede escribir un método que compare dos objetos en la aplicación. Ese método se puede usar en un delegado para un algoritmo de ordenación. Como el código de comparación es independiente de la biblioteca, el método de ordenación puede ser más general.

Se han agregado **punteros de función** a C# 9 para escenarios similares, donde se necesita más control sobre la convención de llamadas. El código asociado a un

delegado se invoca mediante un método virtual agregado a un tipo de delegado. Mediante los punteros de función puede especificar otras convenciones.

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](#).
- Las expresiones lambda son una manera más concisa de escribir bloques de código alineado. En determinados contextos, las expresiones lambda se compilan en tipos de delegado. 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: Fundamentals of C# 3.0](#)

Consulte también

- [Delegate](#)
- [Guía de programación de C#](#)
- [Eventos](#)

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto ↗](#)

Utilizar delegados (Guía de programación de C#)

Artículo • 31/07/2023

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 `Callback` que puede encapsular un método que toma una `string` como argumento y devuelve `void`:

C#

```
public delegate void Callback(string message);
```

Normalmente, un objeto delegado se construye al proporcionar el nombre del método que el delegado encapsulará o con una [expresión lambda](#). 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:

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback 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 argumento 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 `De1` como parámetro:

C#

```
public static void MethodWithCallback(int param1, int param2, Callback
callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Luego puede pasar el delegado creado anteriormente a ese método:

C#

```
MethodWithCallback(1, 2, handler);
```

Y recibir los siguientes resultados en la consola:

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 pasarlala 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 firma del delegado. Cuando se crea un delegado para encapsular un método estático, solo hace referencia al método. Considere las siguientes declaraciones:

C#

```
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 encapsular 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:

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback 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:

C#

```
//remove Method1  
allMethodsDelegate -= d1;  
  
// copy AllMethodsDelegate while removing d2  
Callback 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:

C#

```
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:

C#

```
delegate void Callback1();  
delegate void Callback2();  
  
static void method(Callback1 d, Callback2 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);  
}
```

Consulte 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#)

Artículo • 07/04/2023

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:

C#

```
// 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](#).

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.

ⓘ Nota

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.

A partir de C# 10, los grupos de métodos con una única sobrecarga tienen un *tipo natural*. Esto significa que el compilador puede deducir el tipo de valor devuelto y los

tipos de parámetro para el tipo delegado:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Ejemplos

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

C#

```
// 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
*/
```

En el ejemplo siguiente, un delegado se asigna a métodos estáticos y de instancia y devuelve información específica de cada uno.

```
C#  
  
// 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.  
 */
```

Consulte 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#)

Artículo • 13/03/2024

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

C#

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomCallback(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomCallback.
    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.
        CustomCallback 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;
```

```

        // Initialize the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Initialize 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](#)
- [Eventos](#)



Colaborar con nosotros en
[GitHub](#)

El origen de este contenido se
puede encontrar en [GitHub](#),



Comentarios de .NET

.NET es un proyecto de código
abierto. Seleccione un vínculo para
proporcionar comentarios:

donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

- ⌚ Abrir incidencia con la documentación
- ✍ Proporcionar comentarios sobre el producto

Procedimiento Declarar un delegado, crear instancias del mismo y utilizarlo (Guía de programación de C#)

Artículo • 07/04/2023

Puede declarar delegados mediante cualquiera de los métodos siguientes:

- Declare un tipo de delegado y declare un método con una firma coincidente:

C#

```
// 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}");  
}
```

C#

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

- Asigne un grupo de métodos a un tipo delegado:

C#

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

- Declare un método anónimo:

C#

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
    { Console.WriteLine($"Notification received for: {name}"); };
```

- Use una expresión lambda:

C#

```
// 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 `ProcessBookCallback`. 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

C#

```
// 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 ProcessBookCallback(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(ProcessBookCallback 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.

C#

```
public delegate void ProcessBookCallback(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:

C#

```
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:

C#

```
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:

C#

```
processBook(b);
```

Se puede llamar a un delegado de forma sincrónica, como en este ejemplo, o bien de forma asíncrona con los métodos `BeginInvoke` y `EndInvoke`.

Consulte también

- [Guía de programación de C#](#)
- [Eventos](#)
- [Delegados](#)

Cadenas y literales de cadena

Artículo • 11/04/2024

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 finalice 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, aunque se recomienda usar el alias proporcionado `string`, ya que funciona incluso sin `using System;`. 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:

C#

```
// 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);

```

El operador `new` no se usa 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 las cadenas

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.

C#

```

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:

C#

```
string str1 = "Hello ";  
string str2 = str1;  
str1 += "World";  
  
System.Console.WriteLine(str2);  
//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 entre comillas

Los *literales de cadena entre comillas* comienzan y terminan con un solo carácter de comilla doble ("") en la misma línea. Los literales de cadena entre comillas son más adecuados para las cadenas que caben en una sola línea y no incluyen [ninguna secuencia de escape](#). Un literal de cadena entre comillas debe insertar caracteres de escape, como se muestra en el ejemplo siguiente:

C#

```
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 = @"\u201cThe \u00C3olean Harp\u201d, by Samuel Taylor Coleridge";
//Output: "The A\u00e3olean Harp", by Samuel Taylor Coleridge
```

Literales de cadenas literales

Los *literales de cadena textual* son más adecuados para cadenas de varias líneas, cadenas que contienen caracteres de barra diagonal inversa o comillas dobles insertadas. Las cadenas textuales conservan los caracteres de nueva línea como parte del texto de la cadena. 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:

C#

```
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."
```

Literales de cadena sin formato

A partir de C# 11, puede usar *literales de cadena sin formato* para crear más fácilmente cadenas de varias líneas o usar cualquier carácter que requiera secuencias de escape. Los *literales de cadena sin formato* eliminan la necesidad de usar secuencias de escape. Puede escribir la cadena, incluido el formato de espacio en blanco, cómo quiera que aparezca en la salida. El *literal de cadena sin formato*:

- Comienza y termina con una secuencia de al menos tres caracteres de comilla doble (""). Se permite que más de tres caracteres consecutivos inicien y terminen la secuencia con el fin de admitir literales de cadena que contengan tres (o más) caracteres de comilla repetidos.
- Los literales de cadena sin formato de una sola línea requieren los caracteres de comilla de apertura y cierre en la misma línea.

- Los literales de cadena sin formato de varias líneas requieren que los caracteres de comilla de apertura y cierre estén en su propia línea.
- En los literales de cadena sin formato de varias líneas, los espacios en blanco a la izquierda de las comillas de cierre se quitan.

En los ejemplos siguientes se muestran estas reglas:

C#

```
string singleLine = """Friends say "hello" as they pass by."""";
string multiLine = """
    Hello World!" is typically the first program someone writes.
""";
string embeddedXML = """
    <element attr = "content">
        <body style="normal">
            Here is the main text
        </body>
        <footer>
            Excerpts from "An amazing story"
        </footer>
    </element >
""";
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
""";
```

En los ejemplos siguientes se muestran los errores del compilador notificados en función de estas reglas:

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
""";

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
var noOutdenting = """
    A line of text.
```

```
Trying to outdent the second line.  
""";
```

Los dos primeros ejemplos no son válidos porque los literales de cadena sin formato de varias líneas requieren la secuencia de comillas de apertura y cierre en su propia línea. El tercer ejemplo no es válido porque se anula la sangría del texto de la secuencia de comillas de cierre.

Al usar literales de cadena entre comillas o literales de cadena textuales, debe considerar los literales de cadena sin formato al generar texto que incluya caracteres que requieran [secuencias de escape](#). Los literales de cadena sin formato serán más fáciles para todos, ya que se parecerán más al texto de salida. Por ejemplo, considere el código siguiente que incluye una cadena de JSON con formato:

C#

```
string jsonString = """  
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "TemperatureCelsius": 25,  
    "Summary": "Hot",  
    "DatesAvailable": [  
        "2019-08-01T00:00:00-07:00",  
        "2019-08-02T00:00:00-07:00"  
    ],  
    "TemperatureRanges": {  
        "Cold": {  
            "High": 20,  
            "Low": -10  
        },  
        "Hot": {  
            "High": 60,  
            "Low": 20  
        }  
    },  
    "SummaryWords": [  
        "Cool",  
        "Windy",  
        "Humid"  
    ]  
}  
""";
```

Compare ese texto con el texto equivalente de nuestro ejemplo de [serialización JSON](#), que no usa esta nueva característica.

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
\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")

⚠️ Advertencia

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 "í", que es el punto de código U+00A1. Sin embargo, si el carácter siguiente es "A" o "a", la secuencia de escape se interpretará entonces como `\xA1A` y producirá "竦", 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.

⚠ Nota

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 de 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.

C#

```
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.
```

A partir de C# 10, se puede utilizar la interpolación de cadenas para inicializar una cadena constante cuando todas las expresiones utilizadas para los marcadores de posición son también cadenas constantes.

A partir de C# 11, puede combinar *literales de cadena sin formato* con interpolaciones de cadenas. La cadena de formato se inicia y termina con tres o más comillas dobles sucesivas. Si la cadena de salida debe contener el carácter { o }, puede usar caracteres \$ adicionales para especificar cuántos caracteres { y } comienzan y terminan una interpolación. Todas las secuencias de menos caracteres { o } se incluye en la salida. En el ejemplo siguiente se muestra cómo puede usar esa característica para mostrar la distancia de un punto desde el origen y colocar el punto entre llaves:

C#

```
int X = 2;  
int Y = 3;  
  
var pointMessage = $$""The point {{X}}, {{Y}} is {{Math.Sqrt(X * X + Y *  
Y)}} from the origin."";  
  
Console.WriteLine(pointMessage);  
// Output:  
// The point {2, 3} is 3.605551275463989 from the origin.
```

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.

C#

```
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 [Aplicación de 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 en realidad una cadena nueva y no modifica la cadena original. Para más información, consulte [Cómo: Buscar cadenas y Procedimiento para modificar el contenido de cadenas](#).

C#

```
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:

C#

```
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:

C#

```
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:

C#

```
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. En los ejemplos siguientes se muestran algunos casos en que una referencia a una cadena nula provoca y no provoca una excepción:

C#

```

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:

C#

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal  
pet");  
sb[0] = 'C';  
System.Console.WriteLine(sb.ToString());  
//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:

C#

```
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
```

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](#).

Artículos relacionados

- En [Modificación del contenido de las cadenas](#) se muestran técnicas para transformar las cadenas y modificar su contenido.
- En [Comparación de cadenas](#) se muestra cómo realizar comparaciones ordinales y culturales específicas de las cadenas.
- En [Concatenación de varias cadenas](#) se muestran diversas formas de combinar varias cadenas en una.
- [Análisis de cadenas mediante String.Split](#): contiene ejemplos de código que muestran cómo usar el método [String.Split](#) para analizar cadenas.

- En [Búsqueda de cadenas](#) se explica cómo usar la búsqueda con texto o patrones específicos de las cadenas.
 - En [Determinación de si una cadena representa un valor numérico](#) se muestra cómo analizar de forma segura una cadena para ver si tiene un valor numérico válido.
 - En [Interpolación de cadenas](#) se describe la característica de interpolación de cadenas que proporciona una sintaxis adecuada para dar formato a las cadenas.
 - En [Operaciones básicas de cadenas](#) se proporcionan vínculos a artículos que usan los métodos `System.String` y `System.Text.StringBuilder` para realizar operaciones básicas de cadenas.
 - En [Análisis de cadenas](#) se describe cómo convertir representaciones de cadenas de tipos base de .NET en instancias de los tipos correspondientes.
 - En [Análisis de cadenas de fecha y hora en .NET](#) se muestra cómo convertir una cadena, como "01/24/2008", en un objeto `System.DateTime`.
 - En [Comparación de cadenas](#) se incluye información sobre cómo comparar cadenas y se proporcionan ejemplos de C# y Visual Basic.
 - En [Uso de la clase StringBuilder](#) se describe cómo crear y modificar objetos de cadena dinámicos con la clase `StringBuilder`.
 - En [LINQ y cadenas](#) se proporciona información sobre cómo realizar varias operaciones de cadena utilizando consultas LINQ.
-

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto ↗](#)

Procedimiento Determinar si una cadena representa un valor numérico (Guía de programación de C#)

Artículo • 08/04/2023

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.

C#

```
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.

ⓘ Nota

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`.

C#

```
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#)

Artículo • 11/04/2024

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.

```
C#  
  
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.
```

ⓘ Nota

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.

C#

```
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:

C#

```
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 que va a asignar el descriptor de acceso `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](#)

Comentarios

¿Le ha resultado útil esta página?

 Sí

 No

[Proporcionar comentarios sobre el producto](#) ↗

Uso de indizadores (Guía de programación de C#)

Artículo • 05/11/2024

Los indizadores son una comodidad sintáctica que le permiten crear una [clase](#), [estructura](#) o [interfaz](#) a la que las aplicaciones cliente pueden acceder como una matriz. El compilador genera una propiedad `Item` (o una propiedad con otro nombre si está presente [IndexerNameAttribute](#)) y los métodos de descriptor de acceso adecuados. Los indexadores se implementan con más frecuencia en tipos cuyo propósito principal consiste en encapsular una matriz o colección interna. Por ejemplo, imagine 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 = tempRecord[4]` en lugar de como `float temp = tempRecord.temps[4]`. La notación del indizador no solo simplifica la sintaxis para las aplicaciones cliente; también hace que la clase y su finalidad sean más intuitivas para que otros desarrolladores las entiendan.

Para declarar un indizador en una clase o un struct, use la palabra clave [this](#), como en este ejemplo:

```
C#  
  
// Indexer declaration  
public int this[int index]  
{  
    // get and set accessors  
}
```

ⓘ Importante

Al declarar un indizador, se generará automáticamente una propiedad denominada `Item` en el objeto. No se puede acceder directamente a la propiedad `Item` desde la instancia de [expresión de acceso a miembros](#). Además, si agrega una propiedad `Item` propia a un objeto con un indizador, obtendrá un [error del compilador CS0102](#). Para evitar este error, use el [IndexerNameAttribute](#) para cambiar de nombre el indizador como se detalla más adelante en este artículo.

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 indizador no se clasifica como una variable; por lo tanto, un valor de indizador no se puede pasar por referencia (como un parámetro `ref` o `out`) a menos que su valor sea una referencia (es decir, se devuelve por referencia).

Para proporcionar el indizador con un nombre que puedan usar otros lenguajes, use [System.Runtime.CompilerServices.IndexerNameAttribute](#), como se muestra en el ejemplo siguiente:

C#

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Este indizador tiene el nombre `TheItem`, ya que lo reemplaza el atributo de nombre del indizador. De forma predeterminada, el nombre del indizador es `Item`.

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]`.

C#

```
public class TempRecord
{
```

```

// Array of temperature values
float[] temps =
[
    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 => temps.Length;

// Indexer declaration.
// If index is out of range, the temps array will throw the exception.
public float this[int index]
{
    get => temps[index];
    set => temps[index] = value;
}

```

Tenga en cuenta que, cuando se evalúa el acceso de un indexador (por ejemplo, en una instrucción `Console.WriteLine`), 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.

C#

```

var 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++)
{
    Console.WriteLine($"Element #{i} = {tempRecord[i]}");
}

```

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 indizador. Este tipo de indexador podría implementarse al buscar la cadena de la colección y devolver el valor adecuado. Como los descriptores de acceso se pueden sobrecargar, pueden coexistir las versiones de cadena y entero.

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.

```
C#  
  
// 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 ArgumentOutOfRangeException(  
            nameof(day),  
            $"Day {day} is not supported.\nDay input must be in the form  
\"Sun\", \"Mon\", etc");  
    }  
}
```

Ejemplo de consumo 2

```
C#  
  
var week = new DayCollection();  
Console.WriteLine(week["Fri"]);  
  
try  
{  
    Console.WriteLine(week["Made-up day"]);  
}  
catch (ArgumentOutOfRangeException e)  
{  
    Console.WriteLine($"Not supported input: {e.Message}");  
}
```

Ejemplo 3

En el ejemplo siguiente se declara una clase que almacena los días de la semana mediante la enumeración `System.DayOfWeek`. Un descriptor de acceso `get` toma un valor `DayOfWeek`, el nombre de un día, y devuelve el entero correspondiente. Por ejemplo, `DayOfWeek.Sunday` devuelve 0, `DayOfWeek.Monday` devuelve 1, y así sucesivamente.

```
C#  
  
using Day = System.DayOfWeek;  
  
class DayOfWeekCollection  
{  
    Day[] days =  
    [  
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,  
        Day.Thursday, Day.Friday, Day.Saturday  
    ];  
  
    // Indexer with only a get accessor with the expression-bodied  
    definition:  
    public int this[Day day] => FindDayIndex(day);  
  
    private int FindDayIndex(Day day)  
    {  
        for (int j = 0; j < days.Length; j++)  
        {  
            if (days[j] == day)  
            {  
                return j;  
            }  
        }  
        throw new ArgumentOutOfRangeException(  
            nameof(day),  
            $"Day {day} is not supported.\nDay input must be a defined  
System.DayOfWeek value.");  
    }  
}
```

Ejemplo de consumo 3

```
C#  
  
var week = new DayOfWeekCollection();  
Console.WriteLine(week[DayOfWeek.Friday]);  
  
try  
{
```

```
        Console.WriteLine(week[(DayOfWeek)43]);
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine($"Not supported input: {e.Message}");
    }
}
```

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 artículo, la clase TempRecord proporciona una propiedad Length que permite al código de cliente comprobar la entrada antes de pasarla al indizador. 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](#).

Consulte también

- [Indizadores](#)
- [Propiedades](#)

Indizadores en interfaces (Guía de programación de C#)

Artículo • 05/11/2024

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:

```
C#  
  
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.

```
C#  
  
// 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 IndexOutOfRangeException exception.
        get => arr[index];
        set => arr[index] = value;
    }
}

```

C#

```

IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// 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

C#

```
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:

C#

```
string IEmployee.this[int index]
{
}
```

Implementa el indizador en la interfaz `IEmployee`, mientras que la siguiente declaración:

C#

```
string ICitizen.this[int index]
{
}
```

Implementa el indizador en la interfaz `ICitizen`.

Consulte también

- [Indizadores](#)
- [Propiedades](#)
- [Interfaces](#)

Comparación entre propiedades e indexadores (Guía de programación de C#)

Artículo • 26/10/2024

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.

 Expandir tabla

Propiedad.	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 <code>get</code> de una propiedad no tiene parámetros.	Un descriptor de acceso <code>get</code> de un indexador tiene la misma lista de parámetros formales que el indexador.
Un descriptor de acceso <code>set</code> de una propiedad contiene el parámetro <code>value</code> implícito.	Un descriptor de acceso <code>set</code> de un indexador tiene la misma lista de parámetros formales que el indexador, y también para el parámetro <code>value</code> .
Admite la sintaxis abreviada con propiedades implementadas automáticamente.	Admite miembros de cuerpo de expresión para obtener solo indexadores.

Consulte también

- [Indizadores](#)
- [Propiedades](#)

Eventos (Guía de programación de C#)

Artículo • 11/04/2024

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, 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: Fundamentals of C# 3.0](#)

Consulte también

- [EventHandler](#)
- [Guía de programación de C#](#)
- [Delegados](#)
- [Crear controladores de eventos en Windows Forms](#)

Comentarios

¿Le ha resultado útil esta página?



Sí



No

[Proporcionar comentarios sobre el producto ↗](#)

Procedimiento Suscribir y cancelar la suscripción a eventos (Guía de programación de C#)

Artículo • 10/03/2023

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 ícono **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`.

```
C#  
  
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:

```
C#  
  
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:

C#

```
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.

C#

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

También puede usar una [expresión lambda](#) para especificar un controlador de eventos:

C#

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

Para suscribirse a eventos mediante una función anónima

Si no tiene que cancelar la suscripción a un evento más adelante, puede usar el operador de asignación y suma (`+=`) para asociar una función anónima como controlador de eventos. 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.

C#

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

No podrá cancelar la suscripción a un evento fácilmente si ha usado una función anónima para suscribirse a él. Para cancelar la suscripción en este escenario, regrese al código donde se ha suscrito al evento, almacene la función anónima en una variable de delegado y, después, agregue el delegado al evento. Se recomienda que no use funciones anónimas para suscribirse a eventos si tiene que cancelar la suscripción al evento en el código más adelante. Para obtener más información sobre las funciones anónimas, consulte [Expresiones lambda](#).

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:

C#

```
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 Producir eventos de una clase base en clases derivadas (Guía de programación de C#)

Artículo • 13/03/2024

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.

ⓘ Nota

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

C#

```
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

- Eventos
- Delegados
- Modificadores de acceso
- Crear controladores de eventos en Windows Forms

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte nuestra guía para colaboradores.

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento Implementar eventos de interfaz (Guía de programación de C#)

Artículo • 13/03/2024

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.

C#

```
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 `IDrawingObject` o en `IShape`.

C#

```
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 class 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.
*/

```

Consulte también

- [Eventos](#)
- [Delegados](#)

- Implementación de interfaz explícita
- Procedimiento para generar eventos de una clase base en clases derivadas

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Procedimiento Implementar descriptores de acceso de eventos personalizados (Guía de programación de C#)

Artículo • 10/03/2023

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.

```
C#  
  
event EventHandler IDrawingObject.OnDraw  
{  
    add  
    {  
        lock (objectLock)  
        {  
            PreDrawEvent += value;  
        }  
    }  
    remove  
    {  
        lock (objectLock)  
        {  
            PreDrawEvent -= value;  
        }  
    }  
}
```

```
    PreDrawEvent -= value;  
}  
}  
}
```

Vea también

- [Eventos](#)
- [event](#)

Parámetros de tipos genéricos (Guía de programación de C#)

Artículo • 13/03/2024

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:

C#

```
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](#).

Puede obtener información sobre las convenciones de nomenclatura para los parámetros de tipo genérico en el artículo sobre [convenciones de nomenclatura](#).

Consulte también

- [System.Collections.Generic](#)
- [Genéricos](#)
- [Diferencias entre plantillas de C++ y tipos genéricos de C#](#)



Colaborar con nosotros en

.NET

Comentarios de .NET

GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Restricciones de tipos de parámetros (Guía de programación de C#)

Artículo • 04/08/2024

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 distintos tipos de restricciones:

 Expandir tabla

Restricción	Descripción
<code>where T : struct</code>	El argumento type debe ser un tipo de valor que no acepta valores NULL, que incluye <code>record</code> <code>struct</code> tipos. Para más información sobre los tipos de valor que admiten un valor NULL, consulte Tipos de valor que admiten un valor NULL . Dado que todos los tipos de valor tienen un constructor sin parámetros accesible, declarado o implícito, la restricción <code>struct</code> implica la restricción <code>new()</code> y no se puede combinar con la restricción <code>new()</code> . No puede combinar la restricción <code>struct</code> con la restricción <code>unmanaged</code> .
<code>where T : class</code>	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 acepta valores NULL, <code>T</code> debe ser un tipo de referencia que no acepta valores NULL.
<code>where T : class?</code>	El argumento de tipo debe ser un tipo de referencia, que acepte o no valores NULL. Esta restricción también se aplica a cualquier clase, interfaz, delegado o tipo de matriz, incluidos los registros.
<code>where T : notnull</code>	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, o bien un tipo de valor que no acepta valores NULL.
<code>where T : unmanaged</code>	El argumento de tipo debe ser un tipo no administrado que no acepta valores NULL. La restricción <code>unmanaged</code> implica la restricción <code>struct</code> y no se puede combinar con las restricciones <code>struct</code> ni <code>new()</code> .
<code>where T : new()</code>	El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción <code>new()</code> debe especificarse en último lugar. La restricción <code>new()</code> no se puede combinar con las restricciones <code>struct</code> ni <code>unmanaged</code> .

Restricción	Descripción
<code>where T : <nombre de la clase base></code>	El argumento de tipo debe ser o derivarse de la clase base especificada. En un contexto que acepta valores NULL, <code>T</code> debe ser un tipo de referencia que no acepta valores NULL derivado de la clase base especificada.
<code>where T : <nombre de la clase base?></code>	El argumento de tipo debe ser o derivarse de la clase base especificada. En un contexto que acepta valores NULL, <code>T</code> puede ser un tipo que acepta valores NULL o que no acepta valores NULL derivados de la clase base especificada.
<code>where T : <nombre de la interfaz></code>	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 acepta valores NULL, <code>T</code> debe ser un tipo que no acepta valores NULL que implementa la interfaz especificada.
<code>where T : <nombre de interfaz?></code>	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 acepta valores NULL, <code>T</code> puede ser un tipo de referencia que acepta valores NULL, un tipo de referencia que no acepta valores NULL o un tipo de valor. <code>T</code> no puede ser un tipo de valor que acepta valores NULL.
<code>where T : U</code>	El argumento de tipo proporcionado por <code>T</code> debe ser o se debe derivar del argumento proporcionado para <code>U</code> . En un contexto que acepta valores NULL, si <code>U</code> es un tipo de referencia que no acepta valores NULL, <code>T</code> debe ser un tipo de referencia que no acepta valores NULL. Si <code>U</code> es un tipo de referencia que acepta valores NULL, <code>T</code> puede ser que acepta valores NULL o que no aceptan valores NULL.
<code>where T : default</code>	Esta restricción resuelve la ambigüedad cuando es necesario especificar un parámetro de tipo sin restricciones al invalidar un método o proporcionar una implementación de interfaz explícita. La restricción <code>default</code> implica el método base sin la restricción <code>class</code> o <code>struct</code> . Para obtener más información, vea la propuesta de especificación de la restricción default .
<code>where T : allows ref struct</code>	Esta antirestricción declara que el argumento de tipo para <code>T</code> puede ser un tipo <code>ref struct</code> . El tipo o método genérico debe obedecer las reglas de seguridad de <code>ref</code> para cualquier instancia de <code>T</code> porque podría ser un <code>ref struct</code> .

Algunas restricciones son mutuamente excluyentes y algunas restricciones deben estar en un orden especificado:

- Puede aplicar como máximo una de las restricciones `struct`, `class`, `class?`, `notnull`, y `unmanaged`. Si proporciona alguna de estas restricciones, debe ser la primera restricción especificada para ese parámetro de tipo.
- La restricción de clase base (`where T : Base` O `where T : Base?`) no se puede combinar con ninguna de las restricciones `struct`, `class`, `class?`, `notnull` o `unmanaged`.

- Puede aplicar como máximo una restricción de clase base, de cualquier forma. Si desea admitir el tipo base que acepta valores NULL, use `Base?`.
- No se puede asignar un nombre a la forma que no acepta valores NULL y que aceptan valores NULL de una interfaz como restricción.
- La restricción `new()` no se puede combinar con las restricciones `struct` o `unmanaged`. Si especifica la restricción `new()`, debe ser la última restricción para ese parámetro de tipo. Las antirestricciones, si procede, pueden seguir la restricción `new()`.
- La restricción `default` solo se puede aplicar en implementaciones de interfaz explícitas o invalidaciones. No se puede combinar con las restricciones `struct` o `class`.
- La antirestricción `allows ref struct` no se puede combinar con las restricciones `class` o `class?`.
- La antirestricción `allows ref struct` debe seguir todas las restricciones para ese parámetro de tipo.

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. Las restricciones se aplican al parámetro de tipo cuando la clase o el método genéricos usan cualquier operación en los miembros genéricos más allá de la asignación simple, lo que incluye llamar a los métodos no admitidos por `System.Object`. Por ejemplo, la restricción de clase base indica al compilador que solo los objetos de este tipo o derivados de este tipo pueden reemplazar ese argumento 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.

C#

```
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 IEnumarator<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:

C#

```
class EmployeeList<T> where T : notnull, Employee, IComparable<T>, new()
{
    // ...
    public void AddDefault()
    {
        T t = new T();
        // ...
    }
}
```

Al aplicar la restricción `where T : class`, evite los operadores `==` y `!=` en el parámetro de tipo porque estos operadores solo prueban la identidad de referencia, no para la igualdad de valores. 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 `==`.

C#

```
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 valores, aplique la restricción `where T : IEquatable<T>` o `where T : IComparable<T>` e implemente la interfaz en cualquier clase usada 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:

```
C#  
  
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 se pueden usar porque no hay 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 se compara un parámetro sin enlazar con `null`, la comparación siempre devuelve false si el argumento 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:

```
C#  
  
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:

C#

```
//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`

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. A diferencia de la mayoría de las demás restricciones, si un argumento de tipo infringe la restricción `notnull`, el compilador genera una advertencia en lugar de un error.

La restricción `notnull` tiene efecto solo cuando se usa en un contexto que admite un valor NULL. Si agrega la restricción `notnull` en un contexto en el que se desconocen los valores NULL, el compilador no genera advertencias ni errores para las infracciones de la restricción.

Restricción `class`

La restricción `class` en un contexto que acepta valores NULL 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 argumento de tipo es un tipo de referencia que admite un valor NULL, el compilador genera una advertencia.

Restricción `default`

La incorporación de tipos de referencia que aceptan valores NULL complica el uso de `T?` en un método o tipo genérico. `T?` se puede usar con la restricción `struct` o `class`, pero una de ellas debe estar presente. Cuando se ha usado la restricción `class`, `T?` se refiere al tipo de referencia que acepta valores NULL para `T`. `T?` se puede usar cuando no se aplica ninguna restricción. En ese caso, `T?` se interpreta como `T` para tipos de

valor y los tipos de referencia. Sin embargo, si `T` es una instancia de `Nullable<T>`, `T?` es igual que `T`. En otras palabras, no se convierte en `T??`.

Dado que `T?` ahora se puede usar sin las restricciones `class` o `struct`, pueden surgir ambigüedades en invalidaciones o implementaciones de interfaz explícitas. En ambos casos, la invalidación no incluye las restricciones, pero las hereda de la clase base. Cuando la clase base no aplica las restricciones `class` o `struct`, las clases derivadas deben especificar de algún modo que una invalidación se aplica al método base sin ninguna restricción. El método derivado aplica la restricción `default`. La restricción `default` no aclara ni la restricción `class` ni la `struct`.

Restricción no administrada

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:

C#

```
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

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:

C#

```
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:

C#

```
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 compila. Tanto `first` como `test` son tipos de delegado, pero son tipos de delegado distintos.

Restricciones de enumeración

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.

C#

```
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:

```
C#

enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
C#

var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Los argumentos de tipo implementan la interfaz declarada

Algunos escenarios requieren que un argumento proporcionado para un parámetro de tipo implemente esa interfaz. Por ejemplo:

```
C#

public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
```

```
    static abstract T operator +(T left, T right);
    static abstract T operator -(T left, T right);
}
```

Este patrón permite al compilador de C# determinar el tipo contenedor para los operadores sobrecargados, o algún método `static virtual` o `static abstract`. Proporciona la sintaxis para que los operadores de suma y resta se puedan definir en un tipo contenedor. Sin esta restricción, los parámetros y argumentos deben declararse como interfaz, en lugar de parámetro de tipo:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract IAdditionSubtraction<T> operator +
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    static abstract IAdditionSubtraction<T> operator -
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

La sintaxis anterior requeriría que los implementadores usaran la [implementación de interfaz explícita](#) para esos métodos. Proporcionar la restricción adicional permite a la interfaz definir los operadores en términos de los parámetros de tipo. Los tipos que implementan la interfaz pueden implementar implícitamente los métodos de interfaz.

Permite la estructura de ref

La antirestricción `allows ref struct` declara que el argumento de tipo correspondiente puede ser un tipo `ref struct`. Las instancias de ese parámetro de tipo deben cumplir las reglas siguientes:

- No se puede aplicar la conversión boxing.
- Participa en [reglas de seguridad de ref](#).
- No se pueden usar instancias en las que no se permite un tipo `ref struct`, como los campos `static`.
- Las instancias se pueden marcar con el modificador `scoped`.

La cláusula `allows ref struct` no se hereda. En el código siguiente:

C#

```
class SomeClass<T, S>
    where T : allows ref struct
    where S : T
{
    // etc
}
```

El argumento de `S` no puede ser `ref struct` porque `S` no tiene la cláusula `allows ref struct`.

Un parámetro de tipo que tenga la cláusula `allows ref struct` no se puede usar como argumento de tipo a menos que el parámetro de tipo correspondiente también tenga la cláusula `allows ref struct`. Esta regla se muestra en el ejemplo siguiente:

C#

```
public class Allow<T> where T : allows ref struct
{

}

public class Disallow<T>

{

}

public class Example<T> where T : allows ref struct
{
    private Allow<T> fieldOne; // Allowed. T is allowed to be a ref struct

    private Disallow<T> fieldTwo; // Error. T is not allowed to be a ref struct
}
```

El ejemplo anterior muestra que un argumento de tipo que puede ser un tipo `ref struct` no puede sustituirse por un parámetro de tipo que no puede ser un tipo `ref struct`.

Consulte también

- [System.Collections.Generic](#)
- [Introducción a los genéricos](#)
- [Clases genéricas](#)
- [new \(restricción\)](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Clases genéricas (Guía de programación de C#)

Artículo • 10/05/2023

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. En 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>`); o dejando sin especificar el parámetro de tipo, por ejemplo, cuando se 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:

C#

```
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.

C#

```
//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:

```
C#  
  
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:

```
C#  
  
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:

```
C#  
  
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:

```
C#  
  
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>`.

Consulte 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#)

Artículo • 03/03/2024

A menudo es útil definir interfaces para las clases de colección genéricas o para las clases genéricas que representan elementos de la colección. Para evitar operaciones de conversión boxing y unboxing en tipos de valor, es mejor usar [interfaces genéricas](#), como `IComparable<T>`, en clases genéricas. En la biblioteca de clases de .NET se definen varias interfaces genéricas para usarlas con las clases de colección del espacio de nombres `System.Collections.Generic`. Para más información sobre estas interfaces, consulte [Interfaces genéricas](#).

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>`. Esta restricción 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>`.

C#

```
//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 IComparable<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);
}
}

public class Program
{
    public 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 =
        [
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        ];

        int[] ages = [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:

C#

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Una interfaz puede definir más de un parámetro de tipo, de la siguiente manera:

C#

```
interface IDictionary<K, V>
{
}
```

Las reglas de herencia que se aplican a las clases también se aplican a las interfaces:

C#

```
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 covariantes, lo que significa que solo usan su parámetro de tipo como valor devuelto. En la biblioteca de clases de .NET, `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:

C#

```
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:

C#

```
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](#).

A partir de C# 11, las interfaces pueden declarar miembros `static abstract` o `static virtual`. Las interfaces que declaran miembros `static abstract` o `static virtual` son casi siempre interfaces genéricas. El compilador debe resolver las llamadas a los métodos `static virtual` y `static abstract` en tiempo de compilación. Los métodos `static virtual` y `static abstract` declarados en interfaces no tienen un mecanismo de distribución en tiempo de ejecución análogo a los métodos `virtual` o `abstract` declarados en clases. En su lugar, el compilador usa la información de tipos disponible en tiempo de compilación. Estos miembros se declaran normalmente en interfaces genéricas. Además, la mayoría de las interfaces que declaran métodos `static virtual` o `static abstract` declaran que uno de los parámetros de tipo debe [implementar la interfaz declarada](#). A continuación, el compilador usa los argumentos de tipo proporcionados para resolver el tipo del miembro declarado.

Consulte también

- [Guía de programación de C#](#)
- [Introducción a los genéricos](#)
- [interface](#)
- [Genéricos](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Métodos genéricos (guía de programación de C#)

Artículo • 29/03/2024

Un método genérico es un método que se declara con parámetros de tipo, de la manera siguiente:

C#

```
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:

C#

```
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:

C#

```
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:

C#

```
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.

C#

```
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>`.

C#

```
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:

C#

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

También puede usar el parámetro de tipo como tipo de valor devuelto de un método. En el ejemplo de código siguiente se muestra un método que devuelve una matriz de tipo `T`:

C#

```
T[] Swap<T>(T a, T b)
{
    return [b, a];
}
```

Especificación del lenguaje C#

Para obtener más información, consulte la [Especificación del lenguaje C#](#).

Consulte también

- [System.Collections.Generic](#)
- [Introducción a los genéricos](#)
- [Métodos](#)

 Colaborar con nosotros en
GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Genéricos y matrices (Guía de programación de C#)

Artículo • 13/03/2024

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.

```
C#  
  
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();  
    }  
}
```

Consulte también

- [System.Collections.Generic](#)
- [Genéricos](#)
- [Matrices](#)
- [Genéricos](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Delegados genéricos (Guía de programación de C#)

Artículo • 07/04/2023

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:

C#

```
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:

C#

```
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.

C#

```
class Stack<T>
{
    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:

C#

```
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`.

C#

```
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)
    {
        if (StackEvent is not null)
            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;
}
```

Consulte 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#)

Artículo • 13/03/2024

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 [Genéricos en 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 <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 admite estos operadores. C# no permite esto; las únicas construcciones de lenguaje permitidas son las que se pueden deducir de las restricciones.

Consulte también

- [Introducción a los genéricos](#)
- [Templates \(Plantillas \[C++\]\)](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Genéricos en tiempo de ejecución (guía de programación de C#)

Artículo • 24/04/2024

Cuando se compila un tipo o método genérico en el lenguaje intermedio común (CIL), contiene metadatos que lo identifican como poseedor de parámetros de tipo. La forma en que se usa CIL 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 CIL. 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:

```
C#  
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>`:

```
C#  
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 CIL. 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 CIL. 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`:

C#

```
class Customer { }
class Order { }
```

C#

```
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 llenarán más tarde. Suponga que la línea siguiente de código crea una pila de otro tipo de referencia, que se denomina `Order`:

C#

```
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`:

C#

```
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.

Consulte también

- [System.Collections.Generic](#)
- [Introducción a los genéricos](#)
- [Genéricos](#)

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Referencia del lenguaje C#

La referencia del lenguaje proporciona una referencia informal a la sintaxis y las expresiones de C# para principiantes y desarrolladores experimentados de C# y .NET.

Referencia del lenguaje C#

INFORMACIÓN GENERAL

[Estrategia del lenguaje C#](#)

REFERENCIA

[Palabras clave de C#](#)

[Operadores y expresiones de C#](#)

[Configurar la versión del lenguaje](#)

[Especificación del lenguaje C#: borrador de C# 8 en curso](#)

Novedades

NOVEDADES

[Novedades de C# 12](#)

[Novedades de C# 11](#)

[Novedades de C# 10](#)

REFERENCIA

[Cambios importantes en el compilador de C#](#)

[Compatibilidad de versiones](#)

Mantente en contacto

REFERENCIA

[Comunidad de desarrolladores de .NET ↗](#)

[YouTube](#)

[Twitter](#)

Especificaciones

Lea la especificación detallada del lenguaje C# y las especificaciones detalladas de las características más recientes.

Especificaciones ECMA y características más recientes

INFORMACIÓN GENERAL

[Proceso de especificación](#)

[Contenido detallado de la especificación ECMA](#)

REFERENCIA

[Especificaciones de las características más recientes](#)

Borrador de especificación ECMA de C#: material introductorio

REFERENCIA

[Prólogo](#)

[Introducción](#)

REFERENCIA

[Ámbito](#)

[Referencias de normativas](#)

[Términos y definiciones](#)

[Descripción general](#)

[Conformidad](#)

Borrador de especificación ECMA de C#: especificación del lenguaje

REFERENCIA

[Estructura léxica](#)

[Conceptos básicos](#)

[Tipos](#)

[variables](#)

[Conversiones](#)

[Patrones](#)

[Expresiones](#)

[Instrucciones](#)



[REFERENCIA](#)

[Espacios de nombres](#)

[Clases](#)

[Estructuras](#)

[Matrices](#)

[Interfaces](#)

[Enumeraciones](#)

[Delegados](#)

[Excepciones](#)

[Atributos](#)

[Código no seguro](#)

Borrador de especificación ECMA de C#: anexos



[REFERENCIA](#)

[Gramática](#)

[Problemas de portabilidad](#)

[Biblioteca estándar](#)

[Comentarios de documentación](#)

[Bibliografía](#)