Table of Contents

Un paseo por el lenguaje C#

03/10/2017 • 5 min to read • Edit Online

C# (pronunciado "si sharp" en inglés) es un lenguaje de programación sencillo, moderno, orientado a objetos y con seguridad de tipos. C# tiene sus raíces en la familia de lenguajes C, y a los programadores de C, C++, Java y JavaScript les resultará familiar inmediatamente.

C# es un lenguaje orientado a objetos, pero también incluye compatibilidad para programación ***orientada a componentes***. El diseño de software contemporáneo se basa cada vez más en componentes de software en forma de paquetes independientes y autodescriptivos de funcionalidad. La clave de estos componentes es que presentan un modelo de programación con propiedades, métodos y eventos; tienen atributos que proporcionan información declarativa sobre el componente; e incorporan su propia documentación. C# proporciona construcciones de lenguaje para admitir directamente estos conceptos, por lo que se trata de un lenguaje muy natural en el que crear y usar componentes de software.

Varias características de C# ayudan en la construcción de aplicaciones sólidas y duraderas: la ***recolección de elementos no utilizados*** automáticamente reclama la memoria ocupada por objetos no utilizados y no accesibles; el ***control de excepciones*** proporciona un enfoque estructurado y extensible para la detección de errores y la recuperación; y el diseño del lenguaje ***con seguridad de tipos*** hace imposible leer desde las variables sin inicializar, indizar matrices más allá de sus límites o realizar conversiones de tipos no comprobados.

C# tiene un ***sistema de tipo unificado***. Todos los tipos de C#, incluidos los tipos primitivos como `int` y `double`, se heredan de un único tipo `object` raíz. Por lo tanto, todos los tipos comparten un conjunto de operaciones comunes, y los valores de todos los tipos se pueden almacenar, transportar y utilizar de manera coherente. Además, C# admite tipos de valor y tipos de referencia definidos por el usuario, lo que permite la asignación dinámica de objetos, así como almacenamiento en línea de estructuras ligeras.

Para asegurarse de que las programas y las bibliotecas de C# pueden evolucionar a lo largo del tiempo de manera compatible, se ha puesto mucho énfasis en el ***versionamiento*** del diseño de C#. Muchos lenguajes de programación prestan poca atención a este problema y, como resultado, los programas escritos en dichos lenguajes se interrumpen con más frecuencia de lo necesario cuando se introducen nuevas versiones de las bibliotecas dependientes. Los aspectos del diseño de C# afectados directamente por las consideraciones de versionamiento incluyen los modificadores `virtual` y `override` independientes, las reglas para la resolución de sobrecargas de métodos y la compatibilidad para declaraciones explícitas de miembros de interfaz.

Hola a todos

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

```
using System;
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Normalmente, los archivos de código fuente de C# tienen la extensión de archivo `.cs`. Suponiendo que el programa "Hola mundo" se almacena en el archivo `hello.cs`, el programa podría compilarse con la línea de

comandos:

```
csc hello.cs
```

que genera un ensamblado ejecutable denominado hello.exe. La salida que genera la aplicación cuando se ejecuta es:

```
Hello, World
```

IMPORTANTE

El comando `csc` compila el marco de trabajo completo y puede no estar disponible en todas las plataformas.

El programa "Hola mundo" empieza con una directiva `using` que hace referencia al espacio de nombres `System`. Los espacios de nombres proporcionan un método jerárquico para organizar las bibliotecas y los programas de C#. Los espacios de nombres contienen tipos y otros espacios de nombres; por ejemplo, el espacio de nombres `System` contiene varios tipos, como la clase `Console` a la que se hace referencia en el programa, y otros espacios de nombres, como `IO` y `Collections`. Una directiva `using` que hace referencia a un espacio de nombres determinado permite el uso no calificado de los tipos que son miembros de ese espacio de nombres. Debido a la directiva `using`, puede utilizar el programa `Console.WriteLine` como abreviatura de `System.Console.WriteLine`.

La clase `Hello` declarada por el programa "Hola mundo" tiene un miembro único, el método llamado `Main`. El método `Main` se declara con el modificador static. Mientras que los métodos de instancia pueden hacer referencia a una instancia de objeto envolvente determinada utilizando la palabra clave `this`, los métodos estáticos funcionan sin referencia a un objeto determinado. Por convención, un método estático denominado `Main` sirve como punto de entrada de un programa.

La salida del programa la genera el método `WriteLine` de la clase `Console` en el espacio de nombres `System`. Esta clase la proporcionan las bibliotecas de clase estándar, a las que, de forma predeterminada, el compilador hace referencia automáticamente.

Hay mucha más información sobre C#. Los temas siguientes proporcionan introducciones a los elementos del lenguaje C#. Estas introducciones proporcionarán información básica sobre todos los elementos del lenguaje y ofrecerán los detalles necesarios para profundizar más en los elementos del lenguaje C#:

- Estructura del programa
  - Conozca los principales conceptos organizativos del lenguaje C#: ***programas***, ***espacios de nombres***, ***tipos***, ***miembros*** y ***ensamblados***.
- Tipos y variables
  - Obtenga información sobre los ***tipos de valor***, los ***tipos de referencia*** y las ***variables*** del lenguaje C#.
- Expresiones
  - Las ***expresiones*** se construyen con ***operandos*** y ***operadores***. Las expresiones producen un valor.
- Instrucciones
  - Use ***instrucciones*** para expresar las acciones de un programa.
- Clases y objetos
  - Las ***clases*** son los tipos más fundamentales de C#. Los ***objetos*** son instancias de una clase. Las clases se generan mediante ***miembros***, que también se tratan en este tema.
- Estructuras
  - Las ***estructuras*** son estructuras de datos que, a diferencia de las clases, son tipos de valor.
- Matrices

- Una ***matriz*** es una estructura de datos que contiene un número de variables a las que se accede mediante índices calculados.

- Interfaces
  - Una ***interfaz*** define un contrato que se puede implementar mediante clases y structs. Una interfaz puede contener métodos, propiedades, eventos e indexadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente especifica los miembros que se deben proporcionar mediante clases o structs que implementan la interfaz.
- Enumeraciones
  - Un ***tipo de enumeración*** es un tipo de valor distinto con un conjunto de constantes con nombre.
- Delegados
  - Un ***tipo de delegado*** representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.
- Atributos
  - Los ***atributos*** permiten a los programas especificar información declarativa adicional sobre los tipos, miembros y otras entidades.

SIGUIENTE

Estructura del programa

03/10/2017 • 3 min to read • Edit Online

Los principales conceptos organizativos en C# son ***programas***, ***espacios de nombres***, ***tipos***, ***miembros*** y ***ensamblados***. Los programas de C# constan de uno o más archivos de origen. Los programas declaran tipos, que contienen miembros y pueden organizarse en espacios de nombres. Las clases e interfaces son ejemplos de tipos. Los campos, los métodos, las propiedades y los eventos son ejemplos de miembros. Cuando se compilan programas de C#, se empaquetan físicamente en ensamblados. Normalmente, los ensamblados tienen la extensión de archivo `.exe` o `.dll`, dependiendo de si implementan ***aplicaciones*** o ***bibliotecas***, respectivamente.

En el ejemplo se declara una clase denominada `Stack` en un espacio de nombres llamado `Acme.Collections`:

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data)
        {
            top = new Entry(top, data);
        }

        public object Pop()
        {
            if (top == null)
            {
                throw new InvalidOperationException();
            }
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;
            public Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

El nombre completo de esta clase es `Acme.Collections.Stack`. La clase contiene varios miembros: un campo denominado `top`, dos métodos denominados `Push` y `Pop`, y una clase anidada denominada `Entry`. La clase `Entry` contiene además tres miembros: un campo denominado `next`, un campo denominado `data` y un constructor. Suponiendo que el código fuente del ejemplo se almacene en el archivo `acme.cs`, la línea de comandos

```
csc /t:library acme.cs
```

compila el ejemplo como una biblioteca (código sin un punto de entrada `Main` y genera un ensamblado

denominado `acme.dll`.

> **IMPORTANTE**
> Los ejemplos anteriores usan `csc` como el compilador C# de línea de comandos. Este compilador es un ejecutable de Windows. Para usar C# en otras plataformas, debe emplear las herramientas de .NET Core. El ecosistema .NET Core usa la CLI de `dotnet` para administrar las compilaciones de línea de comandos. Esto incluye la administración de dependencias y la llamada al compilador de C#. Consulte este tutorial para obtener una descripción completa de estas herramientas en las plataformas compatibles con .NET Core.

Los ensamblados contienen código ejecutable en forma de instrucciones de lenguaje intermedio (IL) e información simbólica en forma de metadatos. Antes de ejecutarlo, el código de IL de un ensamblado se convierte automáticamente en código específico del procesador mediante el compilador de Just in Time (JIT) de .NET Common Language Runtime.

Dado que un ensamblado es una unidad autodescriptiva de funcionalidad que contiene código y metadatos, no hay necesidad de directivas `#include` ni archivos de encabezado de C#. Los tipos y miembros públicos contenidos en un ensamblado determinado estarán disponibles en un programa de C# simplemente haciendo referencia a dicho ensamblado al compilar el programa. Por ejemplo, este programa usa la clase `Acme.Collections.Stack` desde el ensamblado `acme.dll`:

```
using System;
using Acme.Collections;
class Example
{
    static void Main()
    {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

Si el programa está almacenado en el archivo `example.cs`, cuando se compila `example.cs`, se puede hacer referencia al ensamblado acme.dl mediante la opción /r del compilador:

```
csc /r:acme.dll example.cs
```

Esto crea un ensamblado ejecutable denominado `example.exe`, que, cuando se ejecuta, produce el resultado:

```
100
10
1
```

C# permite que el texto de origen de un programa se almacene en varios archivos de origen. Cuando se compila un programa de C# de varios archivos, todos los archivos de origen se procesan juntos y los archivos de origen pueden hacerse referencia mutuamente de forma libre; desde un punto de vista conceptual, es como si todos los archivos de origen estuvieran concatenados en un archivo de gran tamaño antes de ser procesados. En C# nunca se necesitan declaraciones adelantadas porque, excepto en contadas ocasiones, el orden de declaración es insignificante. C# no limita un archivo de origen a declarar solamente un tipo público ni precisa que el nombre del archivo de origen coincida con un tipo declarado en el archivo de origen.

ANTERIOR SIGUIENTE

Tipos y variables

03/10/2017 • 7 min to read • Edit Online

Hay dos clases de tipos en C#: *tipos de valor* y *tipos de referencia*. Las variables de tipos de valor contienen directamente los datos, mientras que las variables de los tipos de referencia almacenan referencias a los datos, lo que se conoce como objetos. Con los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y que, por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los tipos de valor, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una variable afecten a la otra (excepto en el caso de las variables de parámetro `ref` y `out`).

Los tipos de valor de C# se dividen en *tipos simples*, *tipos de enumeración*, *tipos de estructura* y *tipos de valores NULL*. Los tipos de referencia de C# se dividen en *tipos de clase*, *tipos de interfaz*, *tipos de matriz* y *tipos delegados*.

A continuación se proporciona información general del sistema de tipos de C#.

- Tipos de valor
  - Tipos simples
    - Entero con signo: `sbyte`, `short`, `int`, `long`
    - Entero sin signo: `byte`, `ushort`, `uint`, `ulong`
    - Caracteres Unicode: `char`
    - Punto flotante de IEEE: `float`, `double`
    - Decimal de alta precisión: `decimal`
    - Booleano: `bool`
  - Tipos de enumeración
    - Tipos definidos por el usuario con el formato `enum E {...}`
  - Tipos de estructura
    - Tipos definidos por el usuario con el formato `struct S {...}`
  - Tipos de valor que aceptan valores NULL
    - Extensiones de todos los demás tipos de valor con un valor `null`
- Tipos de referencia
  - Tipos de clase
    - Clase base definitiva de todos los demás tipos: `object`
    - Cadenas Unicode: `string`
    - Tipos definidos por el usuario con el formato `class C {...}`
  - Tipos de interfaz
    - Tipos definidos por el usuario con el formato `interface I {...}`
  - Tipos de matriz
    - Unidimensional y multidimensional; por ejemplo, `int[]` y `int[,]`
  - Tipos delegados
    - Tipos definidos por el usuario con el formato `delegate int D(...)`

Los ocho tipos enteros proporcionan compatibilidad con valores de 8, 16, 32 y 64 bits en formato con o sin signo.

Los dos tipos de punto flotante, `float` y `double`, se representan mediante los formatos IEC-60559 de precisión sencilla de 32 bits y de doble precisión de 64 bits, respectivamente.

El tipo `decimal` es un tipo de datos de 128 bits adecuado para cálculos financieros y monetarios.

El tipo `bool` de C# se utiliza para representar valores booleanos; valores que son `true` o `false`.

El procesamiento de caracteres y cadenas en C# utiliza la codificación Unicode. El tipo `char` representa una unidad de código UTF-16 y el tipo `string` representa una secuencia de unidades de código UTF-16.

Resume los tipos numéricos de C#.

- Entero con signo
  - `sbyte`: 8 bits, de -128 a 127
  - `short`: 16 bits, de -32,768 a 32,767
  - `int`: 32 bits, de -2,147,483,648 a 2,147,483,647
  - `long`: 64 bits, de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
- Entero sin signo
  - `byte`: 8 bits, de 0 a 255
  - `ushort`: 16 bits, de 0 a 65,535
  - `uint`: 32 bits, de 0 a 4,294,967,295
  - `ulong`: 64 bits, de 0 a 18,446,744,073,709,551,615
- Punto flotante
  - `float`: 32 bits, de 1.5×10^{-45} a 3.4×10^{38}, precisión de 7 dígitos
  - `double`: 64 bits, de 5.0×10^{-324} a 1.7×10^{308}, precisión de 15 dígitos
- Decimal
  - `decimal`: 128 bits, al menos de -7.9×10^{-28} a 7.9×10^{28}, con una precisión mínima de 28 dígitos

Los programas de C# utilizan *declaraciones de tipos* para crear nuevos tipos. Una declaración de tipos especifica el nombre y los miembros del nuevo tipo. Cinco de las categorías de tipos de C# las define el usuario: tipos de clase, tipos de estructura, tipos de interfaz, tipos de enumeración y tipos delegados.

A tipo `class` define una estructura de datos que contiene miembros de datos (campos) y miembros de función (métodos, propiedades y otros). Los tipos de clase admiten herencia única y polimorfismo, mecanismos por los que las clases derivadas pueden extender y especializar clases base.

Un tipo `struct` es similar a un tipo de clase, por el hecho de que representa una estructura con miembros de datos y miembros de función. Sin embargo, a diferencia de las clases, las estructuras son tipos de valor y no suelen requerir la asignación del montón. Los tipos struct no admiten la herencia especificada por el usuario y todos los tipos de struct se heredan implícitamente del tipo `object`.

Un tipo `interface` define un contrato como un conjunto con nombre de miembros de función públicos. Un `class` o `struct` que implementa un `interface` debe proporcionar implementaciones de miembros de función de la interfaz. Un `interface` puede heredar de varias interfaces base, y un `class` o `struct` pueden implementar varias interfaces.

Un tipo `delegate` representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son análogos a los tipos de función proporcionados por los lenguajes funcionales. Son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.

Los tipos `class`, `struct`, `interface` y `delegate` admiten parámetros genéricos, mediante los cuales se pueden parametrizar con otros tipos.

Un tipo `enum` es un tipo distinto con constantes con nombre. Cada tipo `enum` tiene un tipo subyacente, que debe ser uno de los ocho tipos enteros. El conjunto de valores de un tipo `enum` es igual que el conjunto de valores del tipo subyacente.

C# admite matrices unidimensionales y multidimensionales de cualquier tipo. A diferencia de los tipos enumerados anteriormente, los tipos de matriz no tienen que ser declarados antes de usarlos. En su lugar, los tipos de matriz se crean mediante un nombre de tipo entre corchetes. Por ejemplo, `int[]` es una matriz unidimensional de `int`, `int[,]` es una matriz bidimensional de `int` y `int[][]` es una matriz unidimensional de la matriz unidimensional de `int`.

Los tipos de valor NULL tampoco tienen que ser declarados antes de usarlos. Para cada tipo de valor distinto de NULL `T`, existe un tipo de valor NULL correspondiente `T?`, que puede tener un valor adicional, `null`. Por ejemplo, `int?` es un tipo que puede contener cualquier número entero de 32 bits o el valor `null`.

El sistema de tipos de C# está unificado, de tal forma que un valor de cualquier tipo puede tratarse como un `object`. Todos los tipos de C# directa o indirectamente se derivan del tipo de clase `object`, y `object` es la clase base definitiva de todos los tipos. Los valores de tipos de referencia se tratan como objetos mediante la visualización de los valores como tipo `object`. Los valores de tipos de valor se tratan como objetos mediante la realización de *operaciones de conversión boxing* y *operaciones de conversión unboxing*. En el ejemplo siguiente, un valor `int` se convierte en `object` y vuelve a `int`.

```
using System;
class BoxingExample
{
    static void Main()
    {
        int i = 123;
        object o = i;    // Boxing
        int j = (int)o;  // Unboxing
    }
}
```

Cuando se convierte un valor de un tipo de valor al tipo `object`, se asigna una instancia `object`, también denominada "box", para contener el valor, y el valor se copia en dicho box. Por el contrario, cuando se convierte una referencia `object` en un tipo de valor, se comprueba si la referencia `object` es un box del tipo de valor correcto y, si la comprobación es correcta, se copia el valor del box.

El sistema de tipos unificado de C# conlleva efectivamente que los tipos de valor pueden convertirse en objetos "a petición". Debido a la unificación, las bibliotecas de uso general que utilizan el tipo `object` pueden usarse con tipos de referencia y tipos de valor.

Hay varios tipos de *variables* en C#, entre otras, campos, elementos de matriz, variables locales y parámetros. Las variables representan ubicaciones de almacenamiento, y cada variable tiene un tipo que determina qué valores pueden almacenarse en la variable, como se muestra a continuación.

- Tipo de valor distinto a NULL
  - Un valor de ese tipo exacto
- Tipos de valor NULL
  - Un valor `null` o un valor de ese tipo exacto
- objeto
  - Una referencia `null`, una referencia a un objeto de cualquier tipo de referencia o una referencia a un valor de conversión boxing de cualquier tipo de valor
- Tipo de clase
  - Una referencia `null`, una referencia a una instancia de ese tipo de clase o una referencia a una instancia de una clase derivada de ese tipo de clase
- Tipo de interfaz
  - Un referencia `null`, una referencia a una instancia de un tipo de clase que implementa dicho tipo de interfaz o una referencia a un valor de conversión boxing de un tipo de valor que implementa dicho tipo

de interfaz

- Tipo de matriz
  - Una referencia `null`, una referencia a una instancia de ese tipo de matriz o una referencia a una instancia de un tipo de matriz compatible
- Tipo delegado
  - Una referencia `null` o una referencia a una instancia de un tipo delegado compatible

Expresiones

03/10/2017 • 3 min to read • Edit Online

Las *expresiones* se construyen con *operandos* y *operadores*. Los operadores de una expresión indican qué operaciones se aplican a los operandos. Ejemplos de operadores incluyen `+`, `-`, `*`, `/` y `new`. Algunos ejemplos de operandos son literales, campos, variables locales y expresiones.

Cuando una expresión contiene varios operadores, la *precedencia* de los operadores controla el orden en que se evalúan los operadores individuales. Por ejemplo, la expresión `x + y * z` se evalúa como `x + (y * z)` porque el operador `*` tiene mayor precedencia que el operador `+`.

Cuando un operando se encuentra entre dos operadores con la misma precedencia, la *asociatividad* de los operadores controla el orden en que se realizan las operaciones:

- Excepto los operadores de asignación, todos los operadores binarios son *asociativos a la izquierda*, lo que significa que las operaciones se realizan de izquierda a derecha. Por ejemplo, `x + y + z` se evalúa como `(x + y) + z`.
- Los operadores de asignación y el operador condicional (`?:`) son *asociativos a la derecha*, lo que significa que las operaciones se realizan de derecha a izquierda. Por ejemplo, `x = y = z` se evalúa como `x = (y = z)`.

La precedencia y la asociatividad pueden controlarse mediante paréntesis. Por ejemplo, `x + y * z` primero multiplica `y` por `z` y luego suma el resultado a `x`, pero `(x + y) * z` primero suma `x` y `y` y luego multiplica el resultado por `z`.

La mayoría de los operadores se pueden *sobrecargar*. La sobrecarga de operador permite la especificación de implementaciones de operadores definidas por el usuario para operaciones donde uno o ambos operandos son de un tipo de struct o una clase definidos por el usuario.

A continuación se resumen los operadores de C#, con las categorías de operador en orden de precedencia de mayor a menor. Los operadores de la misma categoría tienen la misma precedencia. En cada categoría, hay una lista de expresiones de esa categoría junto con la descripción de ese tipo de expresión.

- Principal
  - `x.m`: acceso a miembros.
  - `x(...)`: invocación de método y delegado.
  - `x[...]`: acceso a matriz e indexador.
  - `x++`: postincremento.
  - `x--`: postdecremento.
  - `new T(...)`: creación de objetos y delegados.
  - `new T(...){...}`: creación de objetos con inicializador.
  - `new {...}`: inicializador de objetos anónimos.
  - `new T[...]`: creación de matriz.
  - `typeof(T)`: obtener el objeto Type para `T`.
  - `checked(x)`: evaluar expresión en contexto comprobado.
  - `unchecked(x)`: evaluar expresión en contexto no comprobado.
  - `default(T)`: obtener valor predeterminado de tipo `T`.
  - `delegate {...}`: función anónima (método anónimo).
- Unario

- `+x`: identidad.
- `-x`: negación.
- `!x`: negación lógica.
- `~x`: negación bit a bit.
- `++x`: preincremento.
- `--x`: predecremento.
- `(T)x`: convertir explícitamente `x` en el tipo `T`.
- `await x`: esperar asincrónicamente a que finalice `x`.

- Multiplicativo
  - `x * y`: multiplicación.
  - `x / y`: división.
  - `x % y`: aviso.
- Aditivo
  - `x + y`: suma, concatenación de cadenas, combinación de delegados.
  - `x – y`: resta, eliminación de delegados.
- Shift
  - `x << y`: desplazamiento a la izquierda.
  - `x >> y`: desplazamiento a la derecha.
- Comprobación de tipos y relacional
  - `x < y`: menor que.
  - `x > y`: mayor que.
  - `x <= y`: menor o igual que.
  - `x >= y`: mayor o igual que.
  - `x is T`: volver a ejecutar `true` si `x` es una `T`, de lo contrario `false`.
  - `x as T`: volver a ejecutar `x` con tipo `T`, o `null` si `x` no es una `T`.
- Igualdad
  - `x == y`: igual que.
  - `x != y`: no igual que.
- AND lógico
  - `x & y`: AND bit a bit entero, AND lógico booleano.
- XOR lógico
  - `x ^ y`: XOR bit a bit entero, XOR lógico booleano.
- OR lógico
  - `x | y`: OR bit a bit entero, OR lógico booleano.
- AND condicional
  - `x && y`: evalúa `y` solo si `x` no es `false`.
- OR condicional
  - `x || y`: evalúa `y` solo si `x` no es `true`.
- Uso combinado de NULL
  - `x ?? y`: se evalúa como `y` si `x` es null, de lo contrario, como `x`.
- Condicional
  - `x ? y : z`: se evalúa como `y` si `x` es `true` o `z` si `x` es `false`.
- Asignación o función anónima
  - `x = y`: asignación.
  - `x op= y`: asignación compuesta; operadores admitidos son:

- `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

- `(T x) => y`: función anónima (expresión lambda).

ANTERIOR SIGUIENTE

Instrucciones

03/10/2017 • 4 min to read • Edit Online

Las acciones de un programa se expresan mediante *instrucciones*. C# admite varios tipos de instrucciones diferentes, varias de las cuales se definen en términos de instrucciones insertadas.

Un *bloque* permite que se escriban varias instrucciones en contextos donde se permite una única instrucción. Un bloque se compone de una lista de instrucciones escritas entre los delimitadores `{` y `}`.

Las *instrucciones de declaración* se usan para declarar variables locales y constantes.

Las *instrucciones de expresión* se usan para evaluar expresiones. Las expresiones que pueden usarse como instrucciones incluyen invocaciones de método, asignaciones de objetos mediante el operador `new`, asignaciones mediante `=` y los operadores de asignación compuestos, operaciones de incremento y decremento mediante los operadores `++` y `--` y expresiones `await`.

Las *instrucciones de selección* se usan para seleccionar una de varias instrucciones posibles para su ejecución en función del valor de alguna expresión. En este grupo están las instrucciones `if` y `switch`.

Las *instrucciones de iteración* se usan para ejecutar una instrucción insertada de forma repetida. En este grupo están las instrucciones `while`, `do`, `for` y `foreach`.

Las *instrucciones de salto* se usan para transferir el control. En este grupo están las instrucciones `break`, `continue`, `goto`, `throw`, `return` y `yield`.

La instrucción `try` ... `catch` se usa para detectar excepciones que se producen durante la ejecución de un bloque, y la instrucción `try` ... `finally` se usa para especificar el código de finalización que siempre se ejecuta, tanto si se ha producido una excepción como si no.

Las instrucciones `checked` y `unchecked` sirven para controlar el contexto de comprobación de desbordamiento para conversiones y operaciones aritméticas de tipo integral.

La instrucción `lock` se usa para obtener el bloqueo de exclusión mutua para un objeto determinado, ejecutar una instrucción y, luego, liberar el bloqueo.

La instrucción `using` se usa para obtener un recurso, ejecutar una instrucción y, luego, eliminar dicho recurso.

A continuación se enumeran los tipos de instrucciones que se pueden usar y se proporciona un ejemplo de cada una.

- Declaración de variable local:

```
static void Declarations(string[] args)
{
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

- Declaración de constante local:

```
static void ConstantDeclarations(string[] args)
{
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

- Instrucción de expresión:

```
static void Expressions(string[] args)
{
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
    Console.WriteLine(i);   // Expression statement
}
```

- Instrucción `if`:

```
static void IfStatement(string[] args)
{
    if (args.Length == 0)
    {
        Console.WriteLine("No arguments");
    }
    else
    {
        Console.WriteLine("One or more arguments");
    }
}
```

- Instrucción `switch`:

```
static void SwitchStatement(string[] args)
{
    int n = args.Length;
    switch (n)
    {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine($"{n} arguments");
            break;
    }
}
```

- Instrucción `while`:

```
static void WhileStatement(string[] args)
{
    int i = 0;
    while (i < args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

- Instrucción `do`:

```
static void DoStatement(string[] args)
{
    string s;
    do
    {
        s = Console.ReadLine();
        Console.WriteLine(s);
    } while (!string.IsNullOrEmpty(s));
}
```

- Instrucción `for`:

```
static void ForStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine(args[i]);
    }
}
```

- Instrucción `foreach`:

```
static void ForEachStatement(string[] args)
{
    foreach (string s in args)
    {
        Console.WriteLine(s);
    }
}
```

- Instrucción `break`:

```
static void BreakStatement(string[] args)
{
    while (true)
    {
        string s = Console.ReadLine();
        if (string.IsNullOrEmpty(s))
            break;
        Console.WriteLine(s);
    }
}
```

- Instrucción `continue`:

```
static void ContinueStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        if (args[i].StartsWith("/"))
            continue;
        Console.WriteLine(args[i]);
    }
}
```

- Instrucción `goto`:

```
static void GoToStatement(string[] args)
{
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length)
        goto loop;
}
```

- Instrucción `return`:

```
static int Add(int a, int b)
{
    return a + b;
}
static void ReturnStatement(string[] args)
{
   Console.WriteLine(Add(1, 2));
   return;
}
```

- Instrucción `yield`:

```
static IEnumerable<int> Range(int from, int to)
{
    for (int i = from; i < to; i++)
    {
        yield return i;
    }
    yield break;
}
static void YieldStatement(string[] args)
{
    foreach (int i in Range(-10,10))
    {
        Console.WriteLine(i);
    }
}
```

- Instrucciones `throw` e instrucciones `try`:

```
static double Divide(double x, double y)
{
    if (y == 0)
        throw new DivideByZeroException();
    return x / y;
}
static void TryCatch(string[] args)
{
    try
    {
        if (args.Length != 2)
        {
            throw new InvalidOperationException("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        Console.WriteLine("Good bye!");
    }
}
```

- Instrucciones `checked` y `unchecked`:

```
static void CheckedUnchecked(string[] args)
{
    int x = int.MaxValue;
    unchecked
    {
        Console.WriteLine(x + 1);  // Overflow
    }
    checked
    {
        Console.WriteLine(x + 1);  // Exception
    }
}
```

- Instrucción `lock`:

```
class Account
{
    decimal balance;
    private readonly object sync = new object();
    public void Withdraw(decimal amount)
    {
        lock (sync)
        {
            if (amount > balance)
            {
                throw new Exception(
                    "Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

- Instrucción `using`:

```
static void UsingStatement(string[] args)
{
    using (TextWriter w = File.CreateText("test.txt"))
    {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

ANTERIOR SIGUIENTE

Clases y objetos

03/10/2017 • 25 min to read • Edit Online

Las *clases* son los tipos más fundamentales de C#. Una clase es una estructura de datos que combina estados (campos) y acciones (métodos y otros miembros de función) en una sola unidad. Una clase proporciona una definición para *instancias* creadas dinámicamente de la clase, también conocidas como *objetos*. Las clases admiten *herencia* y *polimorfismo*, mecanismos por los que las *clases derivadas* pueden extender y especializar *clases base*.

Las clases nuevas se crean mediante declaraciones de clase. Una declaración de clase se inicia con un encabezado que especifica los atributos y modificadores de la clase, el nombre de la clase, la clase base (si se indica) y las interfaces implementadas por la clase. Al encabezado le sigue el cuerpo de la clase, que consta de una lista de declaraciones de miembros escritas entre los delimitadores `{` y `}`.

La siguiente es una declaración de una clase simple denominada `Point`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Las instancias de clases se crean mediante el operador `new`, que asigna memoria para una nueva instancia, invoca un constructor para inicializar la instancia y devuelve una referencia a la instancia. Las instrucciones siguientes crean dos objetos Point y almacenan las referencias en esos objetos en dos variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

La memoria ocupada por un objeto se reclama automáticamente cuando el objeto ya no es accesible. No es necesario ni posible desasignar explícitamente objetos en C#.

Miembros

Los miembros de una clase son miembros estáticos o miembros de instancia. Los miembros estáticos pertenecen a clases y los miembros de instancia pertenecen a objetos (instancias de clases).

A continuación se proporciona una visión general de los tipos de miembros que puede contener una clase.

- Constantes
  - Valores constantes asociados a la clase
- Campos
  - Variables de la clase
- Métodos
  - Cálculos y acciones que pueden realizarse mediante la clase
- Propiedades
  - Acciones asociadas a la lectura y escritura de propiedades con nombre de la clase
- Indizadores

    - Acciones asociadas a la indexación de instancias de la clase como una matriz
- Eventos
    - Notificaciones que puede generar la clase
- Operadores
    - Conversiones y operadores de expresión admitidos por la clase
- Constructores
    - Acciones necesarias para inicializar instancias de la clase o la clase propiamente dicha
- Finalizadores
    - Acciones que deben realizarse antes de que las instancias de la clase se descarten de forma permanente
- Tipos
    - Tipos anidados declarados por la clase

Accesibilidad

Cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa que pueden tener acceso al miembro. Existen cinco formas posibles de accesibilidad. Se resumen a continuación.

- `public`
    - Acceso no limitado
- `protected`
    - Acceso limitado a esta clase o a las clases derivadas de esta clase
- `internal`
    - Acceso limitado al ensamblado actual (.exe, .dll, etc.)
- `protected internal`
    - Acceso limitado a la clase contenedora o a las clases derivadas de la clase contenedora
- `private`
    - Acceso limitado a esta clase

Parámetros de tipo

Una definición de clase puede especificar un conjunto de parámetros de tipo poniendo tras el nombre de clase una lista de nombres de parámetro de tipo entre corchetes angulares. Los parámetros de tipo pueden usarse luego en el cuerpo de las declaraciones de clase para definir a los miembros de la clase. En el ejemplo siguiente, los parámetros de tipo de `Pair` son `TFirst` y `TSecond`:

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Un tipo de clase que se declara para tomar parámetros de tipo se conoce como *tipo de clase genérica*. Los tipos struct, interfaz y delegado también pueden ser genéricos. Cuando se usa la clase genérica, se deben proporcionar argumentos de tipo para cada uno de los parámetros de tipo:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

Un tipo genérico con argumentos de tipo proporcionado, como `Pair<int,string>` anteriormente, se conoce como *tipo construido*.

Clases base

Una declaración de clase puede especificar una clase base colocando después del nombre de clase y los parámetros de tipo dos puntos seguidos del nombre de la clase base. Omitir una especificación de la clase base es igual que derivarla del tipo `object`. En el ejemplo siguiente, la clase base de `Point3D` es `Point` y la clase base de `Point` es `object`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Point3D: Point
{
    public int z;
    public Point3D(int x, int y, int z) :
        base(x, y)
    {
        this.z = z;
    }
}
```

Una clase hereda a los miembros de su clase base. La herencia significa que una clase contiene implícitamente todos los miembros de su clase base, excepto la instancia y los constructores estáticos, y los finalizadores de la clase base. Una clase derivada puede agregar nuevos miembros a aquellos de los que hereda, pero no puede quitar la definición de un miembro heredado. En el ejemplo anterior, `Point3D` hereda los campos `x` y `y` de `Point` y cada instancia de `Point3D` contiene tres campos: `x`, `y` y `z`.

Existe una conversión implícita de un tipo de clase a cualquiera de sus tipos de clase base. Por lo tanto, una variable de un tipo de clase puede hacer referencia a una instancia de esa clase o a una instancia de cualquier clase derivada. Por ejemplo, dadas las declaraciones de clase anteriores, una variable de tipo `Point` puede hacer referencia a una instancia de `Point` o `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Campos

Un *campo* es una variable que está asociada con una clase o a una instancia de una clase.

Un campo declarado con el modificador "static" define un campo estático. Un campo estático identifica exactamente una ubicación de almacenamiento. Independientemente del número de instancias de una clase que se creen, siempre solo hay una copia de un campo estático.

Un campo declarado sin el modificador "static" define un campo de instancia. Cada instancia de una clase contiene una copia independiente de todos los campos de instancia de esa clase.

En el ejemplo siguiente, cada instancia de la clase `Color` tiene una copia independiente de los campos de instancia `r`, `g` y `b`, pero solo hay una copia de los campos estáticos `Black`, `White`, `Red`, `Green` y `Blue`:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;
    public Color(byte r, byte g, byte b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Como se muestra en el ejemplo anterior, los *campos de solo lectura* se puede declarar con un modificador `readonly`. La asignación a un campo `readonly` solo se puede producir como parte de la declaración del campo o en un constructor de la misma clase.

Métodos

Un *método* es un miembro que implementa un cálculo o una acción que puede realizar un objeto o una clase. A los *métodos estáticos* se accede a través de la clase. A los *métodos de instancia* se accede a través de instancias de la clase.

Los métodos pueden tener una lista de *parámetros*, que representan valores o referencias a variables que se pasan al método, y un *tipo de valor devuelto*, que especifica el tipo del valor calculado y devuelto por el método. El tipo de valor devuelto de un método es `void` si no se devuelve un valor.

Al igual que los tipos, los métodos también pueden tener un conjunto de parámetros de tipo, para lo cuales se deben especificar argumentos de tipo cuando se llama al método. A diferencia de los tipos, los argumentos de tipo a menudo se pueden deducir de los argumentos de una llamada al método y no es necesario proporcionarlos explícitamente.

La *signatura* de un método debe ser única en la clase en la que se declara el método. La signatura de un método se compone del nombre del método, el número de parámetros de tipo y el número, los modificadores y los tipos de sus parámetros. La signatura de un método no incluye el tipo de valor devuelto.

Parámetros

Los parámetros se usan para pasar valores o referencias a variables a métodos. Los parámetros de un método obtienen sus valores reales de los *argumentos* que se especifican cuando se invoca el método. Hay cuatro tipos de parámetros: parámetros de valor, parámetros de referencia, parámetros de salida y matrices de parámetros.

Un *parámetro de valor* se usa para pasar argumentos de entrada. Un parámetro de valor corresponde a una variable local que obtiene su valor inicial del argumento que se ha pasado para el parámetro. Las modificaciones en un parámetro de valor no afectan el argumento que se pasa para el parámetro.

Los parámetros de valor pueden ser opcionales; se especifica un valor predeterminado para que se puedan omitir los argumentos correspondientes.

Un *parámetro de referencia* se usa para pasar argumentos mediante una referencia. El argumento pasado para un parámetro de referencia debe ser una variable con un valor definitivo, y durante la ejecución del método, el parámetro de referencia representa la misma ubicación de almacenamiento que la variable del argumento. Un parámetro de referencia se declara con el modificador `ref`. En el ejemplo siguiente se muestra el uso de parámetros `ref`.

```
using System;
class RefExample
{
    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void SwapExample()
    {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine($"{i} {j}");    // Outputs "2 1"
    }
}
```

Un *parámetro de salida* se usa para pasar argumentos mediante una referencia. Es similar a un parámetro de referencia, excepto que no necesita que asigne un valor explícitamente al argumento proporcionado por el autor de la llamada. Un parámetro de salida se declara con el modificador `out`. En el siguiente ejemplo se muestra el uso de los parámetros `out` con la sintaxis que se ha presentado en C# 7.

```
    using System;
    class OutExample
    {
        static void Divide(int x, int y, out int result, out int remainder)
        {
            result = x / y;
            remainder = x % y;
        }
        public static void OutUsage()
        {
            Divide(10, 3, out int res, out int rem);
            Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
        }
    }
}
```

Una *matriz de parámetros* permite que se pasen a un método un número variable de argumentos. Una matriz de parámetros se declara con el modificador `params`. Solo el último parámetro de un método puede ser una matriz de parámetros y el tipo de una matriz de parámetros debe ser un tipo de matriz unidimensional. Los métodos Write y WriteLine de la clase `@System.Console` son buenos ejemplos de uso de la matriz de parámetros. Se declaran de la manera siguiente.

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Dentro de un método que usa una matriz de parámetros, la matriz de parámetros se comporta exactamente igual que un parámetro normal de un tipo de matriz. Sin embargo, en una invocación de un método con una matriz de parámetros, es posible pasar un único argumento del tipo de matriz de parámetros o cualquier número de argumentos del tipo de elemento de la matriz de parámetros. En este caso, una instancia de matriz se e inicializa automáticamente con los argumentos dados. Este ejemplo

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

es equivalente a escribir lo siguiente.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Cuerpo del método y variables locales

El cuerpo de un método especifica las instrucciones que se ejecutarán cuando se invoca el método.

Un cuerpo del método puede declarar variables que son específicas de la invocación del método. Estas variables se denominan *variables locales*. Una declaración de variable local especifica un nombre de tipo, un nombre de variable y, posiblemente, un valor inicial. En el ejemplo siguiente se declara una variable local `i` con un valor inicial de cero y una variable local `j` sin ningún valor inicial.

```
using System;
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}
```

C# requiere que se *asigne definitivamente* una variable local antes de que se pueda obtener su valor. Por ejemplo, si la declaración de `i` anterior no incluyera un valor inicial, el compilador notificaría un error con los usos posteriores de `i` porque `i` no se asignaría definitivamente en esos puntos del programa.

Puede usar una instrucción `return` para devolver el control a su llamador. En un método que devuelve `void`, las instrucciones `return` no pueden especificar una expresión. En un método que devuelve valores distintos de void, las instrucciones `return` deben incluir una expresión que calcula el valor devuelto.

Métodos estáticos y de instancia

Un método declarado con un modificador static es un *método estático*. Un método estático no opera en una instancia específica y solo puede acceder directamente a miembros estáticos.

Un método declarado sin un modificador static es un *método de instancia*. Un método de instancia opera en una instancia específica y puede acceder a miembros estáticos y de instancia. Se puede acceder explícitamente a la instancia en la que se invoca un método de instancia como `this`. Es un error hacer referencia a `this` en un método estático.

La siguiente clase `Entity` tiene miembros estáticos y de instancia.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}
```

Cada instancia `Entity` contiene un número de serie (y probablemente alguna otra información que no se muestra aquí). El constructor `Entity` (que es como un método de instancia) inicializa la nueva instancia con el siguiente número de serie disponible. Dado que el constructor es un miembro de instancia, se le permite acceder al campo de instancia `serialNo` y al campo estático `nextSerialNo`.

Los métodos estáticos `GetNextSerialNo` y `SetNextSerialNo` pueden acceder al campo estático `nextSerialNo`, pero sería un error para ellas acceder directamente al campo de instancia `serialNo`.

En el ejemplo siguiente se muestra el uso de la clase Entity.

```
using System;
class EntityExample
{
    public static void Usage()
    {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());            // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());            // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());    // Outputs "1002"
    }
}
```

Tenga en cuenta que los métodos estáticos `SetNextSerialNo` y `GetNextSerialNo` se invocan en la clase, mientras que el método de instancia `GetSerialNo` se invoca en instancias de la clase.

Métodos virtual, de reemplazo y abstracto

Cuando una declaración de método de instancia incluye un modificador `virtual`, se dice que el método es un *método virtual*. Cuando no existe un modificador virtual, se dice que el método es un *método no virtual*.

Cuando se invoca un método virtual, el *tipo en tiempo de ejecución* de la instancia para la que tiene lugar esa invocación determina la implementación del método real que se invocará. En una invocación de método no virtual, el *tipo en tiempo de compilación* de la instancia es el factor determinante.

Un método virtual puede ser *reemplazado* en una clase derivada. Cuando una declaración de método de instancia incluye un modificador "override", el método reemplaza un método virtual heredado con la misma signatura. Mientras que una declaración de método virtual introduce un método nuevo, una declaración de método de

reemplazo especializa un método virtual heredado existente proporcionando una nueva implementación de ese método.

Un *método abstracto* es un método virtual sin implementación. Un método abstracto se declara con el modificador "abstract" y solo se permite en una clase que también se declare abstracta. Un método abstracto debe reemplazarse en todas las clases derivadas no abstractas.

En el ejemplo siguiente se declara una clase abstracta, `Expression`, que representa un nodo de árbol de expresión y tres clases derivadas, `Constant`, `VariableReference` y `Operation`, que implementan nodos de árbol de expresión para constantes, referencias a variables y operaciones aritméticas. (Esto es similar a los tipos de árbol de expresión, pero no debe confundirse con ellos).

```
using System;
using System.Collections.Generic;
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string,object> vars);
}
public class Constant: Expression
{
    double value;
    public Constant(double value)
    {
        this.value = value;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        return value;
    }
}
public class VariableReference: Expression
{
    string name;
    public VariableReference(string name)
    {
        this.name = name;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        object value = vars[name];
        if (value == null)
        {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}
public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;
    public Operation(Expression left, char op, Expression right)
    {
        this.left = left;
        this.op = op;
        this.right = right;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

Las cuatro clases anteriores se pueden usar para modelar expresiones aritméticas. Por ejemplo, usando instancias de estas clases, la expresión `x + 3` se puede representar de la manera siguiente.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

El método `Evaluate` de una instancia `Expression` se invoca para evaluar la expresión determinada y generar un valor `double`. El método toma un argumento `Dictionary` que contiene nombres de variables (como claves de las entradas) y valores (como valores de las entradas). Como `Evaluate` es un método abstracto, las clases no abstractas que derivan de `Expression` deben invalidar `Evaluate`.

Una implementación de `Constant` de `Evaluate` simplemente devuelve la constante almacenada. Una implementación de `VariableReference` busca el nombre de variable en el diccionario y devuelve el valor resultante. Una implementación de `Operation` evalúa primero los operandos izquierdo y derecho (mediante la invocación recursiva de sus métodos `Evaluate`) y luego realiza la operación aritmética correspondiente.

El siguiente programa usa las clases `Expression` para evaluar la expresión `x * (y + 2)` para los distintos valores de `x` y `y`.

```
using System;
using System.Collections.Generic;
class InheritanceExample
{
    public static void ExampleUsage()
    {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Dictionary<string,object> vars = new Dictionary<string, object>();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));  // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));  // Outputs "16.5"
    }
}
```

Sobrecarga de métodos

La *sobrecarga* de métodos permite que varios métodos de la misma clase tengan el mismo nombre mientras tengan signaturas únicas. Al compilar una invocación de un método sobrecargado, el compilador usa la *resolución de sobrecarga* para determinar el método concreto que de invocará. La resolución de sobrecarga busca el método que mejor coincida con los argumentos o informa de un error si no se puede encontrar ninguna mejor coincidencia. En el ejemplo siguiente se muestra la resolución de sobrecarga en vigor. El comentario para cada invocación del método `Main` muestra qué método se invoca realmente.

```
using System;
class OverloadingExample
{
    static void F()
    {
        Console.WriteLine("F()");
    }
    static void F(object x)
    {
        Console.WriteLine("F(object)");
    }
    static void F(int x)
    {
        Console.WriteLine("F(int)");
    }
    static void F(double x)
    {
        Console.WriteLine("F(double)");
    }
    static void F<T>(T x)
    {
        Console.WriteLine("F<T>(T)");
    }
    static void F(double x, double y)
    {
        Console.WriteLine("F(double, double)");
    }
    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<string>(string)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<int>(int)
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Tal como se muestra en el ejemplo, un método determinado siempre se puede seleccionar mediante la conversión explícita de los argumentos en los tipos de parámetros exactos o el suministro explícito de los argumentos de tipo.

Otros miembros de función

Los miembros que contienen código ejecutable se conocen colectivamente como *miembros de función* de una clase. En la sección anterior se han descrito métodos, que son el tipo principal de los miembros de función. En esta sección se describen los otros tipos de miembros de función admitidos por C#: constructores, propiedades, indexadores, eventos, operadores y finalizadores.

A continuación se muestra una clase genérica llamada List, que implementa una lista creciente de objetos. La clase contiene varios ejemplos de los tipos más comunes de miembros de función.

```
public class List<T>
{
    // Constant
    const int defaultCapacity = 4;

    // Fields
    T[] items;
    int count;

    // Constructor
```

```
public List(int capacity = defaultCapacity)
{
    items = new T[capacity];
}

// Properties
public int Count => count;

public int Capacity
{
    get { return items.Length; }
    set
    {
        if (value < count) value = count;
        if (value != items.Length)
        {
            T[] newItems = new T[value];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
    }
}

// Indexer
public T this[int index]
{
    get
    {
        return items[index];
    }
    set
    {
        items[index] = value;
        OnChanged();
    }
}

// Methods
public void Add(T item)
{
    if (count == Capacity) Capacity = count * 2;
    items[count] = item;
    count++;
    OnChanged();
}
protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as List<T>);

static bool Equals(List<T> a, List<T> b)
{
    if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(b, null) || a.count != b.count)
        return false;
    for (int i = 0; i < a.count; i++)
    {
        if (!object.Equals(a.items[i], b.items[i]))
        {
            return false;
        }
    }
return true;
}

// Event
public event EventHandler Changed;
```

```
    // Operators
    public static bool operator ==(List<T> a, List<T> b) =>
        Equals(a, b);

    public static bool operator !=(List<T> a, List<T> b) =>
        !Equals(a, b);
}
```

Constructores

C# admite constructores de instancia y estáticos. Un *constructor de instancia* es un miembro que implementa las acciones necesarias para inicializar una instancia de una clase. Un *constructor estático* es un miembro que implementa las acciones necesarias para inicializar una clase en sí misma cuando se carga por primera vez.

Un constructor se declara como un método sin ningún tipo de valor devuelto y el mismo nombre que la clase contenedora. Si una declaración de constructor incluye un modificador "static", declara un constructor estático. De lo contrario, declara un constructor de instancia.

Los constructores de instancias se pueden sobrecargar y pueden tener parámetros opcionales. Por ejemplo, la clase `List<T>` declara dos constructores de instancia, una sin parámetros y otra que toma un parámetro `int`. Los constructores de instancia se invocan mediante el operador `new`. Las siguientes instrucciones asignan dos instancias `List<string>` mediante el constructor de la clase `List` con y sin el argumento opcional.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

A diferencia de otros miembros, los constructores de instancia no se heredan y una clase no tiene ningún constructor de instancia que no sea el que se declara realmente en la clase. Si no se proporciona ningún constructor de instancia para una clase, se proporciona automáticamente uno vacío sin ningún parámetro.

Propiedades

Las *propiedades* son una extensión natural de los campos. Ambos son miembros con nombre con tipos asociados y la sintaxis para acceder a los campos y las propiedades es la misma. Sin embargo, a diferencia de los campos, las propiedades no denotan ubicaciones de almacenamiento. Las propiedades tienen *descriptores de acceso* que especifican las instrucciones que se ejecutan cuando se leen o escriben sus valores.

Una propiedad se declara como un campo, salvo que la declaración finaliza con un descriptor de acceso get y un descriptor de acceso set escrito entre los delimitadores `{` y `}` en lugar de finalizar en un punto y coma. Una propiedad que tiene un descriptor de acceso get y un descriptor de acceso set es una *propiedad de lectura y escritura*, una propiedad que tiene solo un descriptor de acceso get es una *propiedad de solo lectura* y una propiedad que tiene solo un descriptor de acceso set es una *propiedad de solo escritura*.

Un descriptor de acceso get corresponde a un método sin parámetros con un valor devuelto del tipo de propiedad. Excepto como destino de una asignación, cuando se hace referencia a una propiedad en una expresión, el descriptor de acceso get de la propiedad se invoca para calcular el valor de la propiedad.

Un descriptor de acceso set corresponde a un método con un solo parámetro denominado value y ningún tipo de valor devuelto. Cuando se hace referencia a una propiedad como el destino de una asignación o como el operando de ++ o--, el descriptor de acceso set se invoca con un argumento que proporciona el nuevo valor.

La clase `List<T>` declara dos propiedades, Count y Capacity, que son de solo lectura y de lectura y escritura, respectivamente. El siguiente es un ejemplo de uso de estas propiedades.

```
List<string> names = new List<string>();
names.Capacity = 100;   // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

De forma similar a los campos y métodos, C# admite propiedades de instancia y propiedades estáticas. Las propiedades estáticas se declaran con el modificador "static", y las propiedades de instancia se declaran sin él.

Los descriptores de acceso de una propiedad pueden ser virtuales. Cuando una declaración de propiedad incluye un modificador `virtual`, `abstract` o `override`, se aplica a los descriptores de acceso de la propiedad.

Indizadores

Un *indexador* es un miembro que permite indexar de la misma manera que una matriz. Un indexador se declara como una propiedad, excepto por el hecho que el nombre del miembro va seguido por una lista de parámetros que se escriben entre los delimitadores `[` y `]`. Los parámetros están disponibles en los descriptores de acceso del indexador. De forma similar a las propiedades, los indexadores pueden ser lectura y escritura, de solo lectura y de solo escritura, y los descriptores de acceso de un indexador pueden ser virtuales.

La clase `List` declara un único indexador de lectura y escritura que toma un parámetro `int`. El indexador permite indexar instancias de `List` con valores `int`. Por ejemplo:

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Los indexadores se pueden sobrecargar, lo que significa que una clase puede declarar varios indexadores siempre y cuando el número o los tipos de sus parámetros sean diferentes.

Eventos

Un *evento* es un miembro que permite que una clase u objeto proporcionen notificaciones. Un evento se declara como un campo, excepto por el hecho de que la declaración incluye una palabra clave event y el tipo debe ser un tipo delegado.

Dentro de una clase que declara un miembro de evento, el evento se comporta como un campo de un tipo delegado (siempre que el evento no sea abstracto y no declare descriptores de acceso). El campo almacena una referencia a un delegado que representa los controladores de eventos que se han agregado al evento. Si no existen controladores de eventos, el campo es `null`.

La clase `List<T>` declara un único miembro de evento llamado `Changed`, lo que indica que se ha agregado un nuevo elemento a la lista. El método virtual `OnChanged` genera el evento cambiado, y comprueba primero si el evento es `null` (lo que significa que no existen controladores). La noción de generar un evento es equivalente exactamente a invocar el delegado representado por el evento; por lo tanto, no hay ninguna construcción especial de lenguaje para generar eventos.

Los clientes reaccionan a los eventos mediante *controladores de eventos*. Los controladores de eventos se asocian mediante el operador `+=` y se quitan con el operador `-=`. En el ejemplo siguiente se asocia un controlador de eventos con el evento `Changed` de un objeto `List<string>`.

```
class EventExample
{
    static int changeCount;
    static void ListChanged(object sender, EventArgs e)
    {
        changeCount++;
    }
    public static void Usage()
    {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);  // Outputs "3"
    }
}
```

Para escenarios avanzados donde se desea controlar el almacenamiento subyacente de un evento, una declaración de evento puede proporcionar explícitamente los descriptores de acceso `add` y `remove`, que son similares en cierto modo al descriptor de acceso `set` de una propiedad.

Operadores

Un *operador* es un miembro que define el significado de aplicar un operador de expresión determinado a las instancias de una clase. Se pueden definir tres tipos de operadores: operadores unarios, operadores binarios y operadores de conversión. Todos los operadores se deben declarar como `public` y `static`.

La clase `List<T>` declara dos operadores, `operator ==` y `operator !=`, y de este modo proporciona un nuevo significado a expresiones que aplican esos operadores a instancias `List`. En concreto, los operadores definen la igualdad de dos instancias `List<T>` como la comparación de cada uno de los objetos contenidos con sus métodos Equals. En el ejemplo siguiente se usa el operador `==` para comparar dos instancias `List<int>`.

```
List<int> a = new List<int>();
a.Add(1);
a.Add(2);
List<int> b = new List<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b);  // Outputs "True"
b.Add(3);
Console.WriteLine(a == b);  // Outputs "False"
```

El primer objeto `Console.WriteLine` genera `True` porque las dos listas contienen el mismo número de objetos con los mismos valores en el mismo orden. Si `List<T>` no hubiera definido `operator ==`, el primer objeto `Console.WriteLine` habría generado `False` porque `a` y `b` hacen referencia a diferentes instancias de `List<int>`.

Finalizadores

Un *finalizador* es un miembro que implementa las acciones necesarias para finalizar una instancia de una clase. Los finalizadores no pueden tener parámetros, no pueden tener modificadores de accesibilidad y no se pueden invocar explícitamente. El finalizador de una instancia se invoca automáticamente durante la recolección de elementos no utilizados.

El recolector de elementos no utilizados tiene una amplia libertad para decidir cuándo debe recolectar objetos y ejecutar finalizadores. En concreto, los intervalos de las llamadas del finalizador no son deterministas y los finalizadores se pueden ejecutar en cualquier subproceso. Por estas y otras razones, las clases deben implementar finalizadores solo cuando no haya otras soluciones que sean factibles.

La instrucción `using` proporciona un mejor enfoque para la destrucción de objetos.

ANTERIOR SIGUIENTE

Estructuras

03/10/2017 • 2 min to read • Edit Online

Al igual que las clases, los ***structs*** son estructuras de datos que pueden contener miembros de datos y miembros de función, pero a diferencia de las clases, los structs son tipos de valor y no requieren asignación del montón. Una variable de un tipo de struct almacena directamente los datos del struct, mientras que una variable de un tipo de clase almacena una referencia a un objeto asignado dinámicamente. Los tipos struct no admiten la herencia especificada por el usuario y todos los tipos de struct se heredan implícitamente del tipo ValueType, que a su vez se hereda implícitamente de `object`.

Los structs son particularmente útiles para estructuras de datos pequeñas que tengan semánticas de valor. Los números complejos, los puntos de un sistema de coordenadas o los pares clave-valor de un diccionario son buenos ejemplos de structs. El uso de un struct en lugar de una clase para estructuras de datos pequeñas puede suponer una diferencia sustancial en el número de asignaciones de memoria que realiza una aplicación. Por ejemplo, el siguiente programa crea e inicializa una matriz de 100 puntos. Si `Point` se implementa como una clase, se crean instancias de 101 objetos distintos: uno para la matriz y uno por cada uno de los 100 elementos.

```
public class PointExample
{
    public static void Main()
    {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++)
            points[i] = new Point(i, i);
    }
}
```

Una alternativa es convertir Point en un struct.

```
struct Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Ahora, se crea la instancia de un solo objeto: la de la matriz, y las instancias de `Point` se asignan en línea dentro de la matriz.

Los structs se invocan con el nuevo operador, pero eso no implica que se asigne memoria. En lugar de asignar dinámicamente un objeto y devolver una referencia a él, un constructor de structs simplemente devuelve el valor del struct propiamente dicho (normalmente en una ubicación temporal en la pila) y este valor se copia luego cuando es necesario.

Con las clases, es posible que dos variables hagan referencia al mismo objeto y, que por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los struct, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una afecten a la otra. Por ejemplo, la salida producida por el fragmento de código siguiente depende de si Point es una clase o un struct.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

Si `Point` es una clase, la salida es 20 porque a y b hacen referencia al mismo objeto. Si Point es un struct, la salida es 10 porque la asignación de `a` a `b` crea una copia del valor, y esta copia no se ve afectada por la asignación posterior a `a.x`.

En el ejemplo anterior se resaltan dos de las limitaciones de los structs. En primer lugar, copiar un struct entero normalmente es menos eficaz que copiar una referencia a un objeto, por lo que el paso de parámetros de asignación y valor puede ser más costoso con structs que con tipos de referencia. En segundo lugar, a excepción de los parámetros `ref` y `out`, no es posible crear referencias a structs, que excluyen su uso en varias situaciones.

Matrices

03/10/2017 • 2 min to read • Edit Online

Una ***matriz*** es una estructura de datos que contiene un número de variables a las que se accede mediante índices calculados. Las variables contenidas en una matriz, denominadas también ***elementos*** de la matriz, son todas del mismo tipo y este tipo se conoce como ***tipo de elemento*** de la matriz.

Los tipos de matriz son tipos de referencia, y la declaración de una variable de matriz simplemente establece un espacio reservado para una referencia a una instancia de matriz. Las instancias de matriz reales se crean dinámicamente en tiempo de ejecución mediante el nuevo operador. La nueva operación especifica la ***longitud*** de la nueva instancia de matriz, que luego se fija para la vigencia de la instancia. Los índices de los elementos de una matriz van de `0` a `Length - 1`. El operador `new` inicializa automáticamente los elementos de una matriz a su valor predeterminado, que, por ejemplo, es cero para todos los tipos numéricos y `null` para todos los tipos de referencias.

En el ejemplo siguiente se crea una matriz de elementos `int`, se inicializa la matriz y se imprime el contenido de la matriz.

```
using System;
class ArrayExample
{
    static void Main()
    {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++)
        {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++)
        {
            Console.WriteLine($"a[{i}] = {a[i]}");
        }
    }
}
```

Este ejemplo se crea y se pone en funcionamiento en una ***matriz unidimensional***. C# también admite ***matrices multidimensionales***. El número de dimensiones de un tipo de matriz, conocido también como ***rango*** del tipo de matriz, es una más el número de comas escritas entre los corchetes del tipo de matriz. En el ejemplo siguiente se asignan una matriz unidimensional, multidimensional y tridimensional, respectivamente.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

La matriz `a1` contiene 10 elementos, la matriz `a2` 50 (10 × 5) elementos y la matriz `a3` 100 (10 × 5 × 2) elementos. El tipo de elemento de una matriz puede ser cualquiera, incluido un tipo de matriz. Una matriz con elementos de un tipo de matriz a veces se conoce como ***matriz escalonada*** porque las longitudes de las matrices de elementos no tienen que ser iguales. En el ejemplo siguiente se asigna una matriz de matrices de `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La primera línea crea una matriz con tres elementos, cada uno de tipo `int[]` y cada uno con un valor inicial de `null`. Las líneas posteriores inicializan entonces los tres elementos con referencias a instancias de matriz individuales de longitud variable.

El nuevo operador permite especificar los valores iniciales de los elementos de matriz mediante un ***inicializador de matriz***, que es una lista de las expresiones escritas entre los delimitadores `{` y `}`. En el ejemplo siguiente se asigna e inicializa un tipo `int[]` con tres elementos.

```
int[] a = new int[] {1, 2, 3};
```

Tenga en cuenta que la longitud de la matriz se deduce del número de expresiones entre { y }. Las declaraciones de variable local y campo se pueden acortar más para que así no sea necesario reformular el tipo de matriz.

```
int[] a = {1, 2, 3};
```

Los dos ejemplos anteriores son equivalentes a lo siguiente:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Interfaces

03/10/2017 • 2 min to read • Edit Online

Una ***interfaz*** define un contrato que se puede implementar mediante clases y structs. Una interfaz puede contener métodos, propiedades, eventos e indexadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente especifica los miembros que se deben proporcionar mediante clases o structs que implementan la interfaz.

Las interfaces pueden usar ***herencia múltiple***. En el ejemplo siguiente, la interfaz `IComboBox` hereda de `ITextBox` y `IListBox`.

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

Las clases y los structs pueden implementar varias interfaces. En el ejemplo siguiente, la clase `EditBox` implementa `IControl` y `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox: IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

Cuando una clase o un struct implementan una interfaz determinada, las instancias de esa clase o struct se pueden convertir implícitamente a ese tipo de interfaz. Por ejemplo

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

En casos donde una instancia no se conoce estáticamente para implementar una interfaz determinada, se pueden usar conversiones de tipo dinámico. Por ejemplo, las siguientes instrucciones usan conversiones de tipo dinámico para obtener las implementaciones de `IControl` y `IDataBound` de un objeto. Dado que el tipo real en tiempo de ejecución del objeto es `EditBox`, las conversiones se realizan correctamente.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

En la clase `EditBox` anterior, el método `Paint` de la interfaz `IControl` y el método `Bind` de la interfaz `IDataBound` se implementan mediante miembros públicos. C# también admite ***implementaciones de miembros de interfaz*** explícitos, lo que permite que la clase o el struct eviten que los miembros se conviertan en públicos. Una implementación de miembro de interfaz explícito se escribe con el nombre de miembro de interfaz completo. Por ejemplo, la clase `EditBox` podría implementar los métodos `IControl.Paint` y `IDataBound.Bind` mediante implementaciones de miembros de interfaz explícitos del modo siguiente.

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() { }
    void IDataBound.Bind(Binder b) { }
}
```

Solo se puede acceder a los miembros de interfaz explícitos mediante el tipo de interfaz. Por ejemplo, la implementación de `IControl.Paint` proporcionada por la clase EditBox anterior solo se puede invocar si se convierte primero la referencia `EditBox` al tipo de interfaz `IControl`.

```
EditBox editBox = new EditBox();
editBox.Paint();            // Error, no such method
IControl control = editBox;
control.Paint();            // Ok
```

Enumeraciones

03/10/2017 • 2 min to read • Edit Online

Un ***tipo de enumeración*** es un tipo de valor distinto con un conjunto de constantes con nombre. Las enumeraciones se definen cuando se necesita fijar un tipo que pueda tener un conjunto de valores discretos. Usan uno de los tipos de valor integral como almacenamiento subyacente. Proporcionan significado semántico a los valores discretos.

En el ejemplo siguiente se declara y usa un tipo `enum` denominado `Color` con tres valores constantes, `Red`, `Green` y `Blue`.

```
using System;
enum Color
{
    Red,
    Green,
    Blue
}
class EnumExample
{
    static void PrintColor(Color color)
    {
        switch (color)
        {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }
    static void Main()
    {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

A cada tipo `enum` le corresponde un tipo entero conocido como el ***tipo subyacente*** del tipo `enum`. Un tipo `enum` que no declara explícitamente un tipo subyacente tiene un tipo subyacente de `int`. El formato de almacenamiento de un tipo `enum` y el intervalo de valores posibles se determinan por el tipo subyacente. El conjunto de valores que un tipo `enum` puede asumir no está limitado por sus miembros `enum`. En concreto, cualquier valor del tipo subyacente de un tipo `enum` puede convertirse en el tipo `enum` y es un valor válido distinto de ese tipo `enum`.

En el ejemplo siguiente se declara un tipo `enum` denominado `Alignment` con un tipo subyacente de `sbyte`.

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

Como se muestra en el ejemplo anterior, una declaración de miembro `enum` puede incluir una expresión constante que especifica el valor del miembro. El valor constante para cada miembro `enum` debe estar en el intervalo del tipo subyacente de `enum`. Cuando una declaración de miembro `enum` no especifica explícitamente un valor, al miembro se le asigna el valor cero (si es el primer miembro en el tipo `enum`) o el valor del miembro textualmente anterior `enum` más uno.

Los valores `Enum` se pueden convertir a valores integrales y viceversa, usando las conversiones de tipo. Por ejemplo:

```
int i = (int)Color.Blue;    // int i = 2;
Color c = (Color)2;         // Color c = Color.Blue;
```

El valor predeterminado de cualquier tipo `enum` es el valor integral cero convertido al tipo `enum`. En los casos donde las variables se inicializan automáticamente en un valor predeterminado, este es el valor que se le asigna a las variables de tipos `enum`. Para que el valor predeterminado de un tipo `enum` esté fácilmente disponible, el literal `0` se convierte implícitamente a cualquier tipo `enum`. Por tanto, el siguiente código es válido.

```
Color c = 0;
```

Delegados

03/10/2017 • 2 min to read • Edit Online

Un ***tipo de delegado*** representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.

En el siguiente ejemplo se declara y usa un tipo de delegado denominado `Function`.

```
using System;
delegate double Function(double x);
class Multiplier
{
    double factor;
    public Multiplier(double factor)
    {
        this.factor = factor;
    }
    public double Multiply(double x)
    {
        return x * factor;
    }
}
class DelegateExample
{
    static double Square(double x)
    {
        return x * x;
    }
    static double[] Apply(double[] a, Function f)
    {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void Main()
    {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

Una instancia del tipo de delegado `Function` puede hacer referencia a cualquier método que tome un argumento `double` y devuelva un valor `double`. El método `Apply` aplica una función dada a los elementos de un argumento `double[]`, y devuelve un valor `double[]` con los resultados. En el método `Main`, `Apply` se usa para aplicar tres funciones diferentes a un valor `double[]`.

Un delegado puede hacer referencia a un método estático (como `Square` o `Math.Sin` en el ejemplo anterior) o un método de instancia (como `m.Multiply` en el ejemplo anterior). Un delegado que hace referencia a un método de instancia también hace referencia a un objeto determinado y, cuando se invoca el método de instancia a través del delegado, ese objeto se convierte en `this` en la invocación.

Los delegados también pueden crearse mediante funciones anónimas, que son "métodos insertados" que se crean sobre la marcha. Las funciones anónimas pueden ver las variables locales de los métodos adyacentes. Por lo tanto, el ejemplo de multiplicador anterior puede escribirse más fácilmente sin utilizarse una clase de Multiplier:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

Una propiedad interesante y útil de un delegado es que no sabe ni necesita conocer la clase del método al que hace referencia; lo único que importa es que el método al que se hace referencia tenga los mismos parámetros y el tipo de valor devuelto que el delegado.

ANTERIOR SIGUIENTE

Atributos

03/10/2017 • 1 min to read • Edit Online

Los tipos, los miembros y otras entidades en un programa de C # admiten modificadores que controlan ciertos aspectos de su comportamiento. Por ejemplo, la accesibilidad de un método se controla mediante los modificadores `public`, `protected`, `internal` y `private`. C # generaliza esta funcionalidad de manera que los tipos de información declarativa definidos por el usuario se puedan adjuntar a las entidades del programa y recuperarse en tiempo de ejecución. Los programas especifican esta información declarativa adicional mediante la definición y el uso de ***atributos***.

En el ejemplo siguiente se declara un atributo `HelpAttribute` que se puede colocar en entidades de programa para proporcionar vínculos a la documentación asociada.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;
    public HelpAttribute(string url)
    {
        this.url = url;
    }

    public string Url => url;

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Todas las clases de atributos se derivan de la clase base Attribute proporcionada por la biblioteca estándar. Los atributos se pueden aplicar proporcionando su nombre, junto con cualquier argumento, entre corchetes, justo antes de la declaración asociada. Si el nombre de un atributo termina en `Attribute`, esa parte del nombre se puede omitir cuando se hace referencia al atributo. Por ejemplo, el atributo `HelpAttribute` se puede usar de la manera siguiente.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/attributes")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/attributes",
    Topic = "Display")]
    public void Display(string text) {}
}
```

En este ejemplo se adjunta un atributo `HelpAttribute` a la clase `Widget`. También se agrega otro atributo `HelpAttribute` al método `Display` en la clase. Los constructores públicos de una clase de atributos controlan la información que se debe proporcionar cuando el atributo se adjunta a una entidad de programa. Se puede proporcionar información adicional haciendo referencia a las propiedades públicas de lectura y escritura de la clase de atributos (como la referencia a la propiedad `Topic` usada anteriormente).

Cuando se solicita un atributo determinado mediante reflexión, se invoca al constructor de la clase de atributos con la información proporcionada en el origen del programa y se devuelve la instancia de atributo resultante. Si se

proporciona información adicional mediante propiedades, dichas propiedades se establecen en los valores dados antes de devolver la instancia del atributo.

ANTERIOR