

Métodos de C#

Artículo • 06/04/2023

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.

ⓘ 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 deben diferenciar en sus tipos de parámetros.

Invocación de método

Los métodos pueden ser de *instancia* o *estáticos*. Para invocar un método de instancia es necesario crear una instancia de un objeto y llamar al método del objeto; el método de una instancia actúa en dicha instancia y sus datos. Si quiere invocar un método estático, haga referencia al nombre del tipo al que pertenece el método; los métodos estáticos no actúan en datos de instancia. Al intentar llamar a un método estático mediante una instancia de objeto se genera un error del compilador.

Llamar a un método es como acceder a un campo. Después del nombre de objeto (si 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` y el segundo 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 a la del método invalidado. El ejemplo siguiente es similar al anterior, salvo que invalida el método `Equals(Object)`. (También invalida el método `GetHashCode()`, ya que los dos métodos están diseñados para proporcionar resultados coherentes).

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 forma predeterminada, los tipos de valor y los tipos de referencia se pasan a un método por valor.

Pasar parámetros por valor

Cuando un tipo de valor se pasa a un método por valor, se pasa una copia del objeto y no el propio objeto. Por lo tanto, los cambios realizados en el objeto en el método llamado no tienen ningún efecto en el objeto original cuando el control vuelve al autor de la llamada.

En el ejemplo siguiente se pasa un tipo de valor a un método por valor, y el método llamado intenta cambiar el valor del tipo de valor. Define una variable de tipo `int`, que es un tipo de valor, inicializa su valor en 20 y lo pasa a un método denominado `ModifyValue` que cambia el valor de la variable a 30. Pero cuando el método vuelve, el valor de la variable no cambia.

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 vuelva 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.

Matrices de parámetros

A veces, el requisito de especificar el número exacto de argumentos al método es restrictivo. El uso de la palabra clave `params` para indicar que un parámetro es una matriz de parámetros permite llamar al método con un número variable de argumentos. El parámetro etiquetado con la palabra clave `params` debe ser un tipo de matriz y ser el último parámetro en la lista de parámetros del método.

Un autor de llamada puede luego invocar el método de una de las tres maneras siguientes:

- Si se pasa una matriz del tipo adecuado que contenga el número de elementos que se quiera.
- Si se pasa una lista separada por comas de los argumentos individuales del tipo adecuado para el método.
- Pasando `null`.
- Si no se proporciona un argumento a la matriz de parámetros.

En el ejemplo siguiente se define un método denominado `GetVowels` que devuelve todas las vocales de una matriz de parámetros. El método `Main` muestra las 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 matriz vacía.

C#

```
static class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(["apple", "banana", "pear"]);
        Console.WriteLine($"Vowels from array: '{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 string[]? input)
    {
        if (input == null || input.Length == 0)
        {
```

```
        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: ''
```

Argumentos y parámetros opcionales

La definición de un método puede especificar que sus parámetros son necesarios o que son opcionales. Los parámetros son necesarios de forma predeterminada. Para especificar parámetros opcionales se incluye el valor predeterminado del parámetro en la definición del método. Cuando se llama al método, si no se proporciona ningún argumento para un parámetro opcional, se usa el valor predeterminado.

El valor predeterminado del parámetro debe asignarse con uno de los siguientes tipos de expresiones:

- Una constante, como una cadena literal o un número.
- Una expresión con el formato `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 acción invoca el constructor sin parámetros implícito del tipo de valor, que no es un miembro real del tipo.



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 parámetros opcionales se definen al final de la lista de parámetros, después de todos los parámetros necesarios.

En el ejemplo siguiente se define un método, `ExampleMethod`, que tiene un parámetro necesario y dos opcionales.

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                             string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} = {re-
quired + optionalInt}";
        Console.WriteLine(msg);
    }
}
```

Si se invoca un método con varios argumentos opcionales mediante argumentos posicionales, el autor de la llamada debe proporcionar un argumento para todos los parámetros opcionales, del primero al último, a los que se proporcione un argumento. Por ejemplo, en el caso del método `ExampleMethod`, si el autor de la llamada proporciona un argumento para el parámetro `description`, también debe proporcionar uno para el parámetro `optionalInt`. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` es una llamada de método válida; `opt.ExampleMethod(2, , "Addition of 2 and 0");` genera un error del compilador, "Falta un argumento".

Si se llama a un método mediante argumentos con nombre o una combinación de argumentos posicionales y con nombre, el autor de la llamada puede omitir los argumentos que siguen al último argumento posicional en la llamada al método.

En el ejemplo siguiente se llama tres veces al método `ExampleMethod`. Las dos primeras llamadas al método usan argumentos posicionales. La primera omite los dos argumentos opcionales, mientras que la segunda omite el último argumento. La tercera llamada 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 incide en la *resolución de sobrecarga* o en la manera en que el compilador de C# determina qué sobrecarga en particular debe invocar una llamada al método, como sigue:

- Un método, indexador o constructor es un candidato para la ejecución si cada uno de sus parámetros es opcional o corresponde, por nombre o por posición, a un solo argumento de la instrucción de llamada y el argumento se puede convertir al tipo del parámetro.
- Si se encuentra más de un candidato, se aplican las reglas de resolución de sobrecarga de las conversiones preferidas a los argumentos que se especifican explícitamente. Los argumentos omitidos en parámetros opcionales se ignoran.
- Si dos candidatos se consideran igualmente correctos, la preferencia pasa a un candidato que no tenga parámetros opcionales cuyos argumentos se hayan omitido en la llamada. Se trata de una consecuencia de una preferencia general en la resolución de sobrecarga para los candidatos con menos parámetros.

Valores devueltos

Los métodos pueden devolver un valor al autor de llamada. Si el tipo de valor devuelto (el tipo que aparece antes del nombre de método) no es `void`, el método puede devolver el valor mediante la palabra clave `return`. Una instrucción con la palabra clave `return` seguida de una variable, una constante o una expresión que coincide con el tipo de valor devuelto devolverá este valor al autor de la llamada al método. Los métodos con un tipo de valor devuelto no nulo son necesarios para usar la palabra clave `return` para devolver un valor. La palabra clave `return` también detiene la ejecución del método.

Si el tipo de valor devuelto es `void`, una instrucción `return` sin un valor también es útil para detener la ejecución del método. Sin la palabra clave `return`, el método dejará de ejecutarse cuando alcance el final del bloque de código.

Por ejemplo, estos dos métodos utilizan la palabra clave `return` para devolver enteros:

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);
```

Usar una variable local, en este caso, `result`, para almacenar un valor es opcional. La legibilidad del código puede ser útil, o puede ser necesaria si debe almacenar el valor original del argumento para todo el ámbito del método.

A veces, quiere que el método devuelva más que un solo valor. Puede hacer esto fácilmente utilizando *tipos de tupla* y *literales de tupla*. El tipo de tupla define los tipos de datos de los elementos de la tupla. Los literales de tupla proporcionan los valores reales de la tupla devuelta. En el ejemplo siguiente, `(string, string, string, int)` define el tipo de tupla que devuelve el método `GetPersonalInfo`. La expresión `(per.FirstName, per.MiddleName, per.LastName, per.Age)` es el literal de tupla; el método devuelve el nombre, los apellidos y la edad de un objeto `PersonInfo`.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

Luego, el autor de la llamada puede usar la tupla devuelta con código como el siguiente:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

También se pueden asignar nombres a los elementos de tupla en la definición de tipo de tupla. En el ejemplo siguiente se muestra una versión alternativa del método `GetPersonalInfo` que usa elementos con nombre:

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("11111111");  
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Si se pasa a un método una matriz como argumento y se modifica el valor de elementos individuales, no es necesario que el método devuelva la matriz, aunque puede que sea preferible hacerlo para conseguir un buen estilo o el flujo funcional de valores. Esto se debe a que C# pasa todos los tipos de referencia por valor, y el valor de una referencia a la matriz es el puntero a la matriz. En el ejemplo siguiente, los cambios al contenido de la matriz `values` que se realizan en el método `DoubleValues` los puede observar cualquier código que tenga una referencia a la matriz.

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. Es obvio que no puede hacerlo si no es propietario del código fuente del tipo. Y esto supone un cambio sustancial si también agrega los campos de datos privados para admitir el método.
- Definir el nuevo método en una clase derivada. No se puede agregar un método de este modo 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`, `IEnumerable<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 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 se trata de un método asíncronico, 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](#).



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](#)