Módulo 1

Programación Orientada a Objetos en .NET Core (C#)

Fundamentos de programación C#

Curso de Programación en .NET

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOi Escuela de organización industrial
Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro
Ayuntamiento de Málaga
pdø de contenidos digitales

Index

Expresiones

Una **expresión** es una forma resumida de implementar una función o método. También se llaman **lambdas** o **funciones flecha**. La forma de construirlas sigue una serie de reglas:

- Después de los parámetros de la función se escribe una flecha (=>).

- Si la función consta de una única instrucción, se pueden omitir las llaves y la palabra return, ya que está implícita. Es decir, la función devuelve lo que devuelva dicha instrucción (si el tipo de la función es void entonces no devuelve nada). En caso de tener más de una única instrucción, se implementa igual que cualquier función, con el cuerpo entre llaves y la palabra return (si es necesario).

- Como veremos más adelante, en el caso de usar tipos delegados, al conocer lo que devuelve la función y los tipos de sus parámetros, esto también se podría omitir. Y si solo recibimos un parámetro también se pueden omitir los paréntesis.

  Ejemplo: Implementación de una función que recibe un parámetro numérico y lo devuelve multiplicado por 2.

  ```
  x => x * 2;
  ```

Constructores y métodos como expresiones

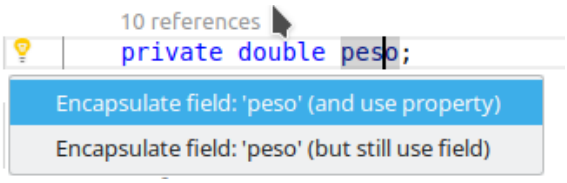
Si el constructor o un método de una clase tienen una única instrucción, se pueden abreviar si los escribimos en formato de expresión. Es decir, en una única línea, sin llaves, y omitiendo (en el caso de que devuelvan algo) la palabra return. Si los métodos son más complejos, no vale la pena hacerlo así, ya que no ahorraríamos nada (sería lo mismo que un método normal con una flecha añadida).

Veamos un ejemplo de una clase simple donde el constructor y sus métodos se han implementado en formato expresión:

```
public class Persona
{
    private string nombre;
    public Persona(string nombre) => this.nombre = nombre;


    public string GetNombre() => nombre;

    public void SetNombre(string nombre) => this.nombre = nombre;

    public void Hablar() => Console.WriteLine($"Hola, me llamo {nombre}");
}
```

Getters/Setters como expresiones

Si queremos crear getters y setters como propiedades públicas, al estilo C#, existe la posibilidad de crearlos a partir de un atributo privado de la clase, situando el cursor encima y haciendo clic en la bombilla de la izquierda, nos ofrece encapsular ese atributo privado creando una propiedad pública (con el nombre en mayúsculas). Se recomienda elegir la primera opción (usar propiedad pública en todos los métodos de la clase), ya que así siempre pasaremos por los filtros del get y el set.



Por defecto esto crea la propiedad con sus get y set como expresiones, ya que son muy simples. Es decir sin las llaves y omitiendo la palabra return.

```
public double Peso { get => peso; set => peso = value; }
```

Funciones delegadas (delegate)

El tipo delegate se utiliza para declarar funciones sin implementar que no pertenecen a una clase. Esto implica que podemos implementarlas de formas diferentes en diversos puntos de nuestro programa, en función de nuestras necesidades.

```
public delegate double OperaNums(double n1, double n2);
```

En este caso hemos declarado un tipo de función llamado OperaNums. Esta función no está implementada, sino que la implementaremos cuando necesitemos utilizarla (pueden haber varias implementaciones diferentes). Lo único que tendremos en cuenta al implementarla es que la función recibirá 2 parámetros de tipo double y deberá devolver un valor double.

Este tipo de datos son candidatos perfectos a implementarlos con una expresión:

```
OperaNums suma = (n1, n2) => n1 + n2;
OperaNums resta = (n1, n2) => n1 - n2;
OperaNums multiplica = (n1, n2) => n1 * n2;
OperaNums divide = (n1, n2) => n1 / n2;

double resultSuma = suma(4, 7);
Console.WriteLine($"4  + 7 = {resultSuma}"); // 11

double resultResta = resta(4, 7);
Console.WriteLine($"4  - 7 = {resultResta}"); // -3

double resultMult = multiplica(4, 7);
Console.WriteLine($"4  * 7 = {resultMult}"); // 28

double resultDivide = divide(4, 7);
Console.WriteLine($"4  / 7 = {resultDivide:f2}"); // 0,57
```

Si nos fijamos, la implementación se guarda en una variable que es del mismo tipo que el delegate. Para llamar a una implementación, simplemente usamos el nombre de la variable como nombre de la función. Esto nos abre posibilidades como guardar funciones dentro de un array o pasarlas por parámetro a un método.

Existen varios tipos de delegates creados en C# que nos permiten crear diferentes funciones. Siempre podemos crearnos los nuestros si así lo necesitamos. Muchos métodos para trabajar con arrays como buscar, etc, admiten como parámetro una función basada en uno de estos tipos de delegates.

Action

El delegate Action representa una función que puede recibir entre 0 y 16 parámetros diferentes, pero no devuelve nada. Útil por ejemplo para mostrar resultados por consola, guardarlos en una base de datos, etc.

Por cada parámetro que va a recibir la función que implementemos , debemos indicarle el tipo entre los símbolos <> (se denominan tipos genéricos). Por ejemplo:

```
// La función recibe una cadena y un número
Action<string, int> longitudMax = (cadena, longitud) => {
    if(cadena.Length <= longitud)
        Console.WriteLine($"'{cadena}' no es mayor de {longitud} caracteres");
    else
        Console.WriteLine($"'{cadena}' es demasiado larga");
};

longitudMax("Hola qué tal", 15); // 'Hola qué tal' no es mayor de 15 caracteres

// La función recibe una lista de enteros y un número

Action<List<int>, int> numDivisibles = (lista, divisor) => {
    int veces = lista.FindAll(n => n % divisor == 0).Count;
    Console.WriteLine($"Hay {veces} números divisibles entre {divisor}");
};

List<int> numeros = new List<int> {16, 36,15,86, 72, 104, 205, 115};
numDivisibles(numeros, 3); // Hay 3 números divisibles entre 3
```

Predicate

El delegate Predicate representa una función que recibe un parámetro de cualquier tipo y comprueba que cumpla unos determinados criterios, devolviendo un booleano que indica si la condición se cumple o no.

```
Predicate<Persona> mayorEdad = p => p.GetEdad() >= 18;
Persona pepe = new Persona("Pepito", 23);
Console.WriteLine(mayorEdad(pepe)); // True
```

Func

El delegate Func representa una función que puede recibir hasta 16 parámetros y que devuelve un resultado. Cuando le indicamos los tipos de datos (igual que en el tipo Action), el último siempre será el del dato que devuelve.

Algunos ejemplos:

```
Func<int, int, int> multiplica = (n1, n2) => n1 * n2;
Func<string, string, int> difLongitud = (s1, s2) => Math.Abs(s1.Length -
s2.Length);
Func<Cuadrado, Cuadrado, Cuadrado> sumaCuadrados = (c1, c2) => new
Cuadrado(c1.GetLado() + c2.GetLado());

Console.WriteLine(multiplica(2, 5)); // 10

Console.WriteLine(difLongitud("caracola", "cebra"));
Cuadrado cu1 = new Cuadrado(4);
Cuadrado cu2 = new Cuadrado(6);
Cuadrado sumaCu = sumaCuadrados(cu1, cu2);
Console.WriteLine($"Lado: {sumaCu.GetLado()}, área: {sumaCu.GetArea()}");
```

Consultas Linq

Las consultas Linq (o Linq queries) son una manera muy versátil y potente de operar con colecciones más allá de los métodos que tienen por defecto. Se basan mucho en las consultas SQL a bases de datos, solo que en este caso se hacen sobre colecciones de datos como listas, arrays, etc.

Los primero que debemos saber es que para usar este tipo de métodos, debemos importar el espacio de nombres **System.Linq**. Esto añadirá una serie de métodos extra a las colecciones como arrays o listas que no estaban antes.

Se pueden plantear este tipo de consultas de forma muy similar a como se hace una consulta SQL a una base de datos, lo que puede ser práctico para quien esté muy familiarizado con dicho lenguaje. Por ejemplo, para obtener los números pares de una lista se podría hacer así:

```
List<int> nums = new List<int> {3, 5, 6, 12, 7, 3, 8, 9, 23, 54};
List<int> pares = (from n in nums
    where (n % 2) == 0
    select n).ToList();

Console.WriteLine(String.Join(",", pares)); // 6, 12, 8, 54
```

Sin embargo, muchos pueden preferir el formato orientado a objetos, es decir, llamar a métodos que van realizando operaciones de filtrado, transformación, etc. Los métodos se pueden consultar en la clase Enumerable y son comunes a las colecciones del lenguaje ya que heredan de dicha clase (esto incluye a los arrays normales).

```
var pares2 = nums.Where(n => n % 2 == 0).ToList();
Console.WriteLine(String.Join(",", pares2));  // 6, 12, 8, 54
```

Estos métodos reciben por parámetro una expresion (normalmente del tipo Action, Predicate o Func). A los métodos que devuelven como resultado una colección (Enumerable), se le pueden seguir concatenando métodos de este tipo.

Lo que tienen en común estos métodos es que no modifican la colección original de datos, siempre devuelven una nueva con cada operación (los que devuelven una colección, claro). Algunos métodos útiles son:

- **All** → Devuelve true solo si todos los elementos cumplen una determinada condición (recibe un Predicate). También tenemos **Any**, que devuelve true simplemente con que uno la cumpla.
- **Concat** → Recibe una colección por parámetro y la concatena a la actual.
- **Average** → Si la colección es numérica, devuelve la media.
- **Distinct** → Elimina los valores repetidos.

- **Except** → Recibe otra colección por parámetro con los valores a excluir de la colección con la que estamos trabajando.

- **First** → Devuelve el primer elemento de la colección. También le podemos pasar un Predicate como parámetro y te devuelve el primer elemento que lo cumple. Cuando no hay resultado que devolver, lanza un error. Si no estamos seguros de si la colección de datos está vacía, podemos usar **FirstOrDefault**, que devuelve null cuando no hay resultados.

- **Last** y **LastOrDefault** → Igual que First y FirstOrDefault pero devuelven el último.

- **Max** y **Min** → Devuelven el valor máximo o mínimo.

- **OrderBy** → Método muy útil para ordenar que recibe un tipo Func por parámetro. Básicamente, consiste en que si tenemos una lista de objetos (generalmente) devolver una propiedad de dicho objeto por la cual se va a ordenar la colección.

- **Enumerable.Range** → Método estático que genera una colección de números automáticamente dentro de un rango. Se especifica el valor mínimo y máximo.

- **Enumerable.Repeat** → Método estático que genera una colección repitiendo un valor N veces. Recibe por parámetro el valor y cuantas copias debe generar.

- **Select** → Aplica una función (Func) de transformación a todos los elementos de la colección, es decir, devuelve una nueva colección con los valores que devuelve la función recibida por parámetro. Opcionalmente, la función que recibe por parámetro puede procesar 2 parámetros, el elemento actual y su posición en la lista (index).

- **Skip** → Quita los N primeros elementos de la colección. Recibe por parámetro cuantos debe saltarse. **SkipLast** hace lo mismo pero con los elementos del final, y **SkipWhile** recibe un Predicate y elimina los elementos que lo cumplan.

- **Sum** → En una colección numérica, devuelve la suma de los elementos.

- **Take** → Devuelve los N primeros elementos (recibe por parámetro cuantos). **TakeLast** hace lo mismo con los N últimos, y **TakeWhile** devuelve los primeros que cumplen la condición del Predicate que recibe por parámetro.

- **ThenBy** → Puesto a continuación de **OrderBy** (y funcionando igual), se utiliza para establecer un segundo campo por el que comparar cuando los campos evaluados por OrderBy son iguales.

- **Where** → Recibe un Predicate y devuelve todos los valores que lo cumplen.

- **ToArray**, **ToList**, **ToDictionary** → Devuelven los resultados como un objeto de la correspondiente colección en lugar de devolver un Enumerable (más limitado).

Existen más métodos útiles. De los que se han omitido, algunos son relativamente complicados de entender al principio, pero no por ello menos interesantes.