# Lambdas en C#

## Índice

# Lambdas en C#

## 1 Introducción

El lenguaje C#, como tantos otros en la década de 2010, ha incorporado varios mecanismos del paradigma de programación funcional[1]. Lenguajes de programación como LISP utilizaban en la década de 1970 las lambdas como una manera de implementar funciones desde un punto de vista más cercano a la definición matemática de función. La razón hay que buscarla en los límites de la minituarización y la llegada de los procesadores multinúcleo. Para el procesamiento paralelo, es muy interesante contar con funciones que no tengan efectos laterales, de manera que puedan lanzarse de manera concurrente.

```
using System;
using System.Collections;

class Lambda1 {
   static void Main()
   {
       Func<int, int> doble = (x) => x * 2;

       Console.WriteLine( $"5 * 2 = {doble( 5 )}" );
   }
}
```

En el código anterior, *doble* es una lambda que calcula el doble de un entero pasado por parámetro. El código anterior sería efectivamente equivalente al siguiente:

```
using System;
using System.Collections;

class Lambda1b {
   static int Doble(int x)
   {
       return x * 2;
   }

   static void Main()
   {
       Console.WriteLine( $"5 * 2 = {Doble( 5 )}" );
   }
}
```

Las ventajas de utilizar lambdas son dos: proporcionan una manera abreviada de escribir una función que no tiene necesidad, además, de estar ligada a una clase; y además, esta función es un valor para un tipo de dato, de manera que es posible que sea pasada por parámetro.

1 Python, C++ y Java han incorporado también lambdas. Como C#, son lenguajes de programación multiparadigma, por lo que la adición no desentona del todo.

# 2 Definición

En la tabla siguiente, se aclaran los tipos que pueden asociarse con las lambdas. Una función siempre devuelve un valor, cuyo tipo $T_{ret}$ se especifica al final de la lista de tipos, mientras que $T_1$ a $T_n$ es la lista de tipos de los parámetros. Los tipos de las acciones (también popularmente conocidas como procedimientos), solo declaran la lista de tipos de los parámetros, de $T_1$ a $T_n$. En ambos casos, la lista de parámetros es opcional, es decir, ni funciones ni acciones tienen necesariamente que tener parámetros.

*Tabla 1: Tipos asociados a una lambda.*

| Tipo | Lambda |
|---|---|
| *Func<[$T_1$ [, $T_2$ [,...$T_n$]]]], $T_{ret}$>* | Función. |
| *Action<[$T_1$ [, $T_2$[,… $T_n$]]]>* | Acción (procedimiento). |

**Sintaxis**

*Func<[$T_1$ [, $T_2$ [,...$T_n$]]]], $T_{ret}$> $f_1$ = ([$Vt_1$ [, $Vt_2$[,… $VT_n$]]]) => expresión_de_tipo_$T_{ret}$;*
*Func<[$T_1$ [, $T_2$ [,...$T_n$]]]], $T_{ret}$> $f_2$ = ([$Vt_1$ [, $Vt_2$[,… $VT_n$]]]) => {*
*instrucción$_1$ ;*
*[… instrucción$_n$;]*
***return** expresión de tipo $T_{ret}$;*
*}*

*Action<[$T_1$ [, $T_2$ [,...$T_n$]]]]> $A_1$ = ([$Vt_1$ [, $Vt_2$[,… $VT_n$]]]) => expresión_de_tipo_$T_{ret}$;*
*Action<[$T_1$ [, $T_2$ [,...$T_n$]]]]> $A_2$ = ([$Vt_1$ [, $Vt_2$[,… $VT_n$]]]) => {*
*instrucción$_1$ ;*
*[… instrucción$_n$;]*
*}*

**Ejemplos**

```
Func<int, int> doble = (x) => x * 2;
Func<int, int, int> suma = (x, y) => x + y;
Func<int, int, int> suma2 = (x, y) => { int toret = x + y; return toret; }

Action<string> out = (x) => Console.WriteLine( x );
Action<string> out2 = (x) => { Console.WriteLine( x ); };
```

Como puede verse tanto en la sintaxis como en los ejemplos anteriores, la forma más sencilla solo define una expresión que será el retorno de la función. Sin embargo, si añadimos llaves podemos emplear un bloque completo. Es la primera forma, la que se concentra en una expresión a devolver, la que más se aproxima a las lambdas que provienen de LISP, pues aunque no esté garantizado en un lenguaje como C#, es la forma en la que se evitan los efectos laterales, y es por tanto en la que nos centraremos.

# 3 Creando *funciones como lambdas*

## *3.1 Recursividad*

Si solo es posible (o al menos nos centraremos en), que una lambda proporcione una expresión como medio de ejecución, no es factible utilizar estructuras de bucles o decisión. La única forma por tanto es emplear, respectivamente, recursividad y el operador ?.

Por ejemplo, el siguiente código calcula una multiplicación de enteros utilizando la suma:

**Ejemplo (erróneo)**

```
Func<int, int, int> multiplicaRec =
            (x, y) => y == 1 ? x : x + multiplicaRec( x, y – 1 );
```

El problema es que, tal y como se muestra el código más arriba, no compila. Esto es debido a que estamos definiendo *multiplicaRec*, pero a la vez, como buena función recursiva, estamos llamando a *multiplicaRec*, que todavía no existe. La única forma de solucionar esto es definir la función previamente, con un valor **null** con el solo objetivo de que la función ya exista previamente para cuando el compilador se encuentre con la llamada recursiva.

**Ejemplo**

```
Func<int, int, int> multiplicaRec = null;
multiplicaRec = (x, y) => y == 1 ? x : x + multiplicaRec( x, y – 1 );
```

## *3.2 El operador ternario*

Un segundo detalle importante es el operador ? o ternario. Este operador se emplea para poder tomar decisiones dentro de una expresión, pues es el equivalente a la estructura *if*, tomando una condición *cond* como primer parámetro, la expresión *expr1* a devolver si la condición resulta ser verdadera, y la expresión expr2 a devolver si la *cond* resulta ser falsa.

**Sintaxis**

*cond ? expr1 : expr2*

**Ejemplo**

```
Console.WriteLine( 5  > 2 ? “5 > 2” : “5 <= 2” );    // Visualiza “5 > 2”

int rndVal = new Random().Next() % 10;
Console.WriteLine( rndVal > 5 ? “Rnd > 5” : “Rnd <= 5” );
```

En el código anterior, el primer *WriteLine()* siempre visualiza “5 > 2”, mientras que lo que visualice el segundo depende del número pseudoaleatorio generado mediante la clase **Random**.

Un programa completo puede verse a continuación, empleando la lambda recursiva más arriba.

```
using System;


class LambdaRecursiva {
   static void Main()
   {
       Func<int, int, int> multiplicaRec = null;
       multiplicaRec =
           (x, y) => y == 1 ? x : x + multiplicaRec( x, y – 1 );

       Console.WriteLine( $”10 * 2 = {multiplicaRec( 5, 2 )}” );
   }
}
```

# 4 Linq

*Linq* fue introducido en C# como una forma de acceder a la información de manera más declarativa, emulando la forma de trabajar de las sentencias SQL. Así, toma la forma de una consulta SQL pero con el orden de las cláusulas ligeramente alterado para poder acomodar el comprobador de tipos.

**Sintaxis**

```
from vble in colección
      [where expr_con_vble]
      [orderby expr2_con_vble]
      select expr3_con_vble;
```

**Ejemplo**

```
int[] v = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> seleccionPares =
                       from x in v
                       where x % 2 == 0
                       select x;         // 2, 4, 6, 8, 10

IEnumerable<int> seleccionDobleDeParesInversa =
                       from x in v
                       where x % 2 == 0
                       orderby x descending
                       select x * 2;     // 20, 16, 12, 8, 4
```

Aunque es posible indicar *var* a la izquierda de *seleccionPares*, aparece el tipo completo (la clase base de todas las colecciones en C#), como referencia. La sentencia *Linq* tiene varias partes: ***from** vble **in** colección*, que es cuando se crea la variable con la que se identificará a cada uno de los miembros individuales de la colección; ***where** x % 2 == 0*, que es opcional y especifica la condición que deben de cumplir los elementos para ser incluídos en la nueva colección; **orderby** x **descending**, que también es opcional especifica el orden de los elementos en la nueva colección: puede ser ***ascending*** (por defecto), o ***descending*** (la inversa); finalmente ***select** x* permite indicar alguna transformación (de existir) que se le quiera aplicar a cada elemento individual antes de

introducirlo en a lista final. Si bien *Linq* soporta otras cláusulas (como ***join***), estas son las más importantes.

Un ejemplo completo se incluye a continuación, en el que se generan varios números sin repeticiones y se filtran los pares en orden descendente:

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


public class Test
{
	public static void Main()
	{
		const int MAX = 20;
		var generados = new HashSet<int>( MAX );
		var lista = new List<int>( MAX );
		var rnd = new Random();

		// Introducir hasta MAX elementos sin repeticiones
		for(int i = 0; i < MAX / 2; ++i) {
			int generado;

			do {
				generado = rnd.Next( MAX + 1 );
			} while( generados.Contains( generado ) );

			generados.Add( generado );
			lista.Add( generado );
		}

		// Seleccionar los pares en orden descendente
		IEnumerable<int> seleccionPares =
									from x in lista
									where x % 2 == 0
									orderby x descending
									select 2 * x;

		// Visualizar
		foreach(int x in seleccionPares) {
			Console.Write( x + " " );
		}

		Console.WriteLine();
	}
}
```

# 5 Linq con lambdas

Existen varios métodos que pueden ser aplicados a las colecciones de C#, como por ejemplo *Select()*, *Where()*, u *OrderBy()*. Estos métodos esperan que les sea pasada una lambda *l* para poder filtrar, proyectar o buscar.

A continuación, los métodos presentes en todas las colecciones (al estar definidos en **IEnumerable**).

- *All(l)*. Devuelve **true** o **false** dependiendo de si todos los elementos cumplen la condición en *l*.
- *Any(l)*. Devuelve **true** o **false** dependiendo de si alguno de los elementos cumple la condición en *l*.
- *FirstOrDefault(l)*. Devuelve el primer elemento de la colección que cumple con la condición en *l*. Si no se encuentra ninguno, devuelve el valor por defecto para el tipo de los elementos de la lista.
- *LastOrDefault(l)*. Devuelve el último elemento de la colección que cumple con la condición en *l*. Si no se encuentra ninguno, devuelve el valor por defecto para el tipo de los elementos de la lista.
- *Count(l)*. Devuelve el número de elementos que cumplen con la condición en *l*. Existe la variante *LongCount(l)* para aquellos casos en los que se espera que la colección sea muy grande y su número de elementos sobrepase **int**.*MaxValue*.
- *Max(l) / Min(l)*. Devuelve el máximo o el mínimo (respectivamente) de aquellos elementos que cumplen con la condición en *l*.
- *Select(l)*. La lambda *l* proporciona una manera de transformar los elementos de una colección en otros, en una nueva lista. Por ejemplo, Select( x => x * 2 ) convierte los elementos de una colección de enteros en su doble. La variante *SelectMany(l)* se emplea para cuando el resultado de cada elemento va a ser una colección a su vez.
- *SingleOrDefault(l)*. Devuelve el único elemento en la colección que cumple la condición en *l*, o bien el valor por defecto para el tipo de la colección si no existe o en cambio, existe más de uno.
- *Sum(l)/Average(l)*. Obtiene la suma (o la media, respectivamente), de aquellos elementos en la colección que cumplen la condición en *l*.
- *OrderBy(l)* / *OrderByDescending(l)*. Ordena los elementos de una lista en una lista resultado. Según se desee ordenar de manera ascendente o descendente, se empleará la primera o la segunda, respectivamente.
- *Where(l)*. Filtra los elementos de una lista a una lista resultado, según el resultado de la lambda *l* (solo cuando la lambda devuelve **true** el elemento es incorporado a la lista resultado).

Sigue la lista de los métodos más relevantes que solo están presentes exclusivamente en **List**.

- *ForEach(l)*. Aplica la lambda *l* a todos los elementos de la lista. Este método no devuelve ningún valor, y *l* a su vez no puede devolver nada. Se trata de realizar una acción sobre todos los elementos, como por ejemplo imprimirlo por pantalla.
- *RemoveAll(l)*. Elimina todos los elementos de la lista que cumplen con la condición en *l*.
- *Exists(l)*. Devuelve **true** o **false** según exista al menos un elemento que cumpla *l*.

Un ejemplo como el de la sección dedicada a Linq, pero en el caso en el que se quiera utilizar Linq con lambdas, es el siguiente:

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


public class Test
{
	public static void Main()
	{
		const int MAX = 20;
		var generados = new HashSet<int>( MAX );
		var lista = new List<int>( MAX );
		var rnd = new Random();

		// Introducir hasta MAX elementos sin repeticiones
		for(int i = 0; i < MAX / 2; ++i) {
			int generado;
			do {
				generado = rnd.Next( MAX + 1 );
			} while( generados.Contains( generado ) );

			generados.Add( generado );
			lista.Add( generado );
		}

		// Seleccionar los pares en orden descendente
		IEnumerable<int> seleccionPares =
					lista
						.Where( x => x % 2 == 0 )
						.OrderByDescending( x => x )
						.Select( x => x * 2 );

		// Visualizar
		foreach(int x in seleccionPares) {
			Console.Write( x + " " );
		}

		Console.WriteLine();
	}
}
```