Programación funcional

C# - Apuntes Tácticos

Índice

1 Expresiones Lambda

1.1 A modo de introducción

Las expresiones lambda son un método sin declaración, es decir, sin modificadores de acceso, que devuelve un valor o puede no devolver nada. Permiten escribir un método y utilizarlo inmediatamente.

Esto es útil en el caso de una única llamada a un método, ya que permite reducir el tiempo de declarar y escribir el método sin tener que crear una clase.

Un método típico está formado por cuatro elementos:

- El tipo que devuelve el método,
- El nombre del método,
- La lista de parámetros, y
- El cuerpo del método donde están las instrucciones que se ejecutan.

```
public int Cuadrado(int num1, int num2)
        {
            return(num1*num2);
        }
```

Las expresiones lambda está formada por tres elementos (no define un nombre de método):

- El tipo que devuelve el método, (si es que lo hay) se infiere del contexto en el que se utiliza la expresión Lambda
- La lista de parámetros, y
- El cuerpo del método donde están las instrucciones que se ejecutan.

```
(num1,num2) =>
       {
            return(num1*num2);
        }
```

1.1.1 Sintaxis

```
( () => {Console.WriteLine(“hola mundo”);} )
() => { Console.WriteLine(“hola mundo”); }
(x) => { return x * x; }
(x, y) => { return x + y; }
(ref x, y) => {
                    x++;
                    return x / y;
} // (Ojo parámetro x pasado por referencia)
```

- () Una lista de parámetros entre paréntesis, si el método que estamos definiendo no acepta parámetro se ponen los paréntesis vacíos
- El operador => que indica que es una expresión Lambda. El operador => tiene la misma prioridad que la asignación (=) y es asociativo por la derecha.
- A la derecha del operador indicaremos el código de la función (la expresión o bloque de instrucciones). El cuerpo del método, puede contener todas las instrucciones que queramos, o únicamente una expresión, y se pueden intentar y/o formatear de la forma que nos parezcan más legibles, solo tenemos que acordarnos de incluir el carácter punto y coma (;) al final de cada linea igual que en un método ordinario
- Si una expresión lambda acepta parámetros, los especificaremos entre paréntesis a la izquierda del operados =>. Podemos omitir el tipo de los parámetros porque el compilador los deducirá del contexto de la expresión Lambda.
- Podemos pasar parámetros por referencia mediante la palabra clave [ref]
- Pueden devolver valores, pero el tipo de retronó se infiere del contexto donde se usa.

1.1.2 Puntos importantes:

- Las expresiones lambda no utilizan la instrucción return, excepto aquellas que utilicen un bloque encerrado entre llaves.

```
suma = (a, b) => { return a + b; };

suma = (a,b)  => a + b;
```

1.1.3 Los delegados genéricos: action, func y predicate

Cuando definimos un delegado lo que estamos haciendo es declarar una variable que apunta a otro método (como un puntero en C y C++). Esto es útil cuando tenemos que pasar una función como parámetro de otra función o de un método

Podemos definir un delegado utilizando la palabra clave delegate seguido de la declaración de la función como vemos a continuación:

```
public delegate int  Operar(int x, int y);
```

Para aprender más sobre delegados

Ejercicio 1: Declarar un delegado que reciba dos enteros y retorne un entero.
Plantear una clase Operacion y los métodos que permitan sumar y restar dos enteros.
Llamar a los métodos mediante la definición de un delegado.

Ejercicio 2: Declarar un delegado que reciba dos enteros y retorne un entero.
Plantear una clase Operacion y los métodos que permitan sumar y restar dos enteros. Un tercer método que reciba un delegado y dos enteros.

Por otro lado, podemos utilizar los siguientes delegados genéricos en lugar de *delegate*. Con ellos conseguimos una sintaxis algo más refinada y simple.

Action se utiliza para aquellas expresiones lambda que no retornan ningún valor.

```
Action<string> saludo = s => Console.Write("Hola {0}!", s);

saludo("Amigo");
```

Func para aquellas expresiones que retornen un valor.

```
Func<int, int, int> suma = (a, b) => a + b;

int resultado = suma(3, 5);
```

En el caso del delegado Func, el último parámetro es un valor de retorno

Predicate similar a los dos anteriores, en este caso, un predicate SIEMPRE devuelve un boolean. Por ejemplo Predicate<int> recibe un int y devuelve un booleano.

```
Predicate<int> mayorDeEdad = e => e >= 18;

bool esMayorDeEdad = mayorDeEdad(10);
```

1.1.4 Lambdas de expresión

Una lambda con una única expresión en el lado derecho se denomina lambda de expresión.

Los paréntesis son opcionales si la expresión lambda tiene un único parámetro de entrada; de lo contrario, son obligatorios. Dos o más parámetros de entrada se separan por comas y se encierran entre paréntesis:

```
(x, y) => x == y
```

A veces, es difícil o imposible para el compilador deducir los tipos de entrada. Cuando esto ocurre, puede especificar los tipos explícitamente como se muestra en el ejemplo siguiente:

```
(int x, string s) => s.Length > x
```

Para especificar cero parámetros de entrada, utilice paréntesis vacíos:

```
() => SomeMethod()
```

Observe en el ejemplo anterior que el cuerpo de una lambda de expresión puede estar compuesto de una llamada a método.

1.1.5 Lambdas de instrucciones

Una lambda de instrucciones es similar a una lambda de expresión, salvo que las instrucciones se encierran entre llaves:

El cuerpo de una lambda de instrucciones puede estar compuesto de cualquier número de instrucciones; sin embargo, en la práctica, generalmente no hay más de dos o tres.

```
(x, y) => {
        if (x == y)
                return true;
        else
                return false;
};
```

1.2 Expresiones Lambda y genéricos

Un ejemplo: La manera natural de implementar una definición para la función lambda cuadrado en C# sería a través de un tipo y una instancia de delegados:

```
delegate T Mapeado<T>(T x);
```

```
Mapeado<int> cuadrado = delegate(int x) { return x * x; };
```

La primera línea del código anterior define un tipo delegado genérico llamado Mapeado. A partir de este modelo podremos crear instancias de delegados para funciones que a partir de un valor de un tipo T producen otro valor del mismo tipo T (o sea, que mapean un valor del tipo T a otro valor del mismo tipo).

En la segunda línea, por otra parte, se define una instancia del tipo delegado anterior que "apunta" a una función que devuelve el cuadrado de un número entero. En esta sintaxis se hace uso de otra característica incorporada a C# en la versión 2.0, los métodos anónimos, que permiten la definición en línea del bloque de código que especifica la funcionalidad a la que se desea asociar a la instancia del delegado.

Ejercicio 3: Declarar un delegado genérico que acepte un tipo y que devuelva ese mismo tipo.
Plantear un método int Cubo(int x) que acepte un entero y que devuelva el triple del mismo ejecutándolo a través del delegado.

Las expresiones lambda pueden considerarse como una extensión de los métodos anónimos, que ofrecen una sintaxis más concisa y funcional para expresarlos. Por ejemplo, nuestra definición de cuadrado quedaría de la siguiente forma utilizando una expresión lambda:

```
Mapeado<int> cuadrado3 = x => x * x;
```

Se expresa de una manera muy sucinta y natural que cuadrado3 hace referencia a una función que, a partir de un x, produce x multiplicado por sí mismo. En este caso, el compilador deduce (infiere) automáticamente del contexto que el tipo del parámetro y del resultado deben ser int. Pudimos haberlo indicado explícitamente:

```
Mapeado<int> cuadrado4 = (int x) => x * x;
```

1.2.1 Un ejemplo

Imaginemos el caso que queremos ordenar una lista, usando el método Sort. El método Sort espera un delegate de tipo Comparison. Este delegate es un delegate genérico que espera dos argumentos de tipo T, y devuelve un int:

```
public delegate int Comparison<T>(T x,T y);
```

Si queremos ordenar una lista, usando un método anónimo, haríamos algo parecido a esto:

```
List<string> lista = new List<string>() { "Perro", "Gato", "Zorro" };
lista.Sort (delegate (string s1, string s2) {
        return s1.CompareTo(s2);
} );
```

Dentro de la llamada del método Sort, creamos el método anónimo (de tipo Comparison<string>).

El código anterior usando expresiones lambda:

```
List<string> lista = new List<string>() { "Perro", "Gato", "Zorro" };
lista.Sort ( (x,y) => x.CompareTo(y));
```

Ejercicio 4: Crea una clase Persona (nombre, apellidos, dni) con un constructor que de valor a los atributos. En la clase donde esté el Main declaras una lista de personas. Esta lista se rellenará con algunas personas.
En la clase donde se encuentra el main debes hacer dos métodos

```
public static List<Persona> OrdenarConDelegado() {}
public static List<Persona> OrdenarConLambda(){}
```

Debes ordenar la lista por apellidos y si son iguales por nombre. Este ejercicio lo deberás hacer de dos formas. La primera utilizando una función anónima (delegate) y la segunda utilizando una expresión lambda.

1.3 Lambdas con los operadores de consulta estándar

1.3.1 Metodos nativos o *extensión methods* para iterar sobre listas

Contains(T)	IEquatable
Equals(T)	IEquatable
Exists(Predicate<T>)	Determina si List<T> contiene elementos que cumplen las condiciones definidas por el predicado
Find(Predicate<T>)	Devuelve el primero que cumple el predicado
FindAll(Predicate<T>)	Devuelve más de uno, si cumple el predicado
ForEach(Action<T>)	Realiza la acción especificada en cada elemento de List<T>.
RemoveAll(Predicate<T>)	Quita todos los elementos que cumplen las condiciones definidas por el predicado especificado.
Sort() comparación como lambda	Implementar IComparable (CompareTo()) o poner la función lambda que compara. Ordena los elementos de toda la List<T> utilizando el comparador predeterminado.
Sort(Comparison<T>)	Ordena los elementos de toda la List<T> utilizando el Comparison<T> especificado.

```
List<string> values = new List<string> { "value1", "value2", "value3" };

values.Exists((v) => v ==  "value1");       //true
values.Find(v => v.Contains("value");       //devolverá el primero
values.FindAll(v => v.Contains("value"));   //devolverá los tres
values.Sort((a,b) => b.CompareTo(a));       //ordena descendente
values.ForEach((v) => Console.WriteLine("Element = "+ v)); //imprime los tres
values.RemoveAll((v) => v.Contains("value"));  //borra los tres
```

2 LINQ en C#

2.1 Qué es LINQ?

Más o menos en 2007 los desarrolladores de C# vieron que tenían cierto problema cuando querían acceder a datos haciendo consultas sobre los mismos y no podían hacerlo de una forma sencilla.

El motivo por el que no se podía hacer de una forma sencilla era porque, había datos en 3 fuentes diferentes, Como son.

- Las colecciones de datos en memoria: para las que necesitamos los diferentes métodos aportados por Generics o diferentes algoritmos para conseguir estos datos.
- Bases de datos: para el que necesitábamos conectores de `ADO.NET` y escribir nuestro propio `SQL`.
- Ficheros XML: Para poder iterar sobre los mismos, necesitábamos utilizar `XmlDocument` o `Xpath`.

Todas estas APIs/librerías tienen diferentes funcionalidades y sintaxis, entonces Microsoft decidió introducir un lenguaje que ofreciera una sintaxis única para todas estas funcionalidades. Y aquí es donde Microsoft introdujo Language Integrated Query o `LINQ`.

`LINQ` nos proporciona **comprobaciones de tipo en consultas durante la compilación**, pero estas consultas van a ser ejecutadas en tiempo de ejecución sobre datos en memoria, contra una base de datos o en xml. Pero además una vez comprendamos LINQ veremos que podemos utilizarlo contra por ejemplo el sistema de archivos de nuestro pc, una base de datos no relacional, ficheros CSV, y mucho más.

2.1.1 Ejemplos de Expresiones Lambda en LINQ

El principal uso de las expresiones lambda está vinculado con las expresiones de consultas LINQ. Así que veamos algunos ejemplos de estas expresiones, estos ejemplos son bien sencillos pero nos vienen bien para comprender el uso de las expresiones lambda.

```
int[] nums = { 3, 4, 5, 6, 4, 5, 7 };
int[] numMayores = nums.Where(n => n > 5).ToArray();
int[] numPares = nums.Where(n => n % 2 == 0).ToArray();
int[] numImpares = nums.Where(n => n % 2 != 0).ToArray();
```

El en código de ejemplo anterior tenemos 3 consultas LINQ:

- La primera consulta la usamos para guardar en la variable de tipo int[] (numMayores) los números que sean mayores de 5, y para lograrlo usamos la expresión lambda (n => n > 5). El resultado sería evidentemente { 6, 7 }.
- En la segunda consulta (numPares) obtendremos los números pares usando la siguiente expresión lambda (n => n % 2 == 0). El resultado sería evidentemente { 4, 6, 4 }.
- En la tercera consulta (numImpares) obtendremos los números impares usando la siguiente expresión lambda (n => n % 2 != 0).

2.1.2 Ejemplos de Expresiones Lambda en LINQ con colecciones de objetos

```
public class Alumno
{
	public int Codigo { get; set; }
	public string Nombre { get; set; }
	public int Edad { get; set; }

	public Alumno(int codigo, string nombre, int edad)
	{
		this.Codigo = codigo;
		this.Nombre = nombre;
		this.Edad = edad;
	}
}
```

```
static void Main(string[] args){
	Console.WriteLine("*** APRENDIENDO LINQ ***");
	List<Alumno> alumnos = new List<Alumno>()
	{
		new Alumno(1, "Gerson", 20),
		new Alumno(2, "Luis", 25),
		new Alumno(3, "Yesica", 17),
		new Alumno(4, "Ana", 28),
		new Alumno(5, "Estefany", 27)
	};
```

Y lo que queremos es *obtener el código de una alumna llamada "Ana"*, veamos:

```
Alumno alu =
(from l in alumnos where l.Nombre == "Ana" select l).FirstOrDefault();
Console.WriteLine(alu.Codigo);
```

El resultado es 4, lo cual es correcto!

Se nos pide ahora mostrar aquellos alumnos que cumplan la siguiente condición: *Tengan 20* años *o más*, codifiquémoslo:

```
List <Alumno> alumnoBuscado2 = (from l in listadoAlumnos where
l.Edad >= 20 select l).ToList();

foreach (Alumno item in alumnoBuscado2)
{
     Console.WriteLine(item.Nombre);
}
Console.ReadLine();
```

Como resultado obtenemos esto:

```
Gerson
Luis
Ana
Estefany
```

2.2 Cómo Utilizar y escribir LINQ

Para realizar este ejemplo, utilizaremos LINQ **sobre datos en memoria**, en este caso un `Array[]` de la clase `Libro`. Podríamos utilizar una `Lista<T>` pero, `List<T>` es específico del namespace `System.Linq` que fue específicamente creado para trabajar con él después de utilizar `IEnumerable`, el cual es el centro de este contenido.

```
public class Libro
{
    public int Id { get; set; }
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public Libro(int id, string titulo, string autor)
    {
        Id = id;
        Titulo = titulo;
        Autor = autor;
    }
}
```

```
Libro[] arrayLibros = new Libro[5];
arrayLibros[0] = new Libro(1, "Poeta en NY", "Federico García Lorca");
```

```
arrayLibros[1] = new Libro(2, "Los asesinos", "S. Posteguillo");
arrayLibros[2] = new Libro(3, "circo máximo", "S. Posteguillo");
arrayLibros[3] = new Libro(4, "Frankenstein", "S. Posteguillo");
arrayLibros[4] = new Libro(5, "El origen perdido", "Matilde Asensi");
```

Antes de empezar a utilizar LINQ hay que tener muy claro y comprender que son los extension methods y sobre todo las expresiones lambda ya que LINQ esta construido a través de estas dos tecnologías principalmente.

Además de hacer una sola API que pudiera consultar todas las fuentes de datos otro objetivo muy importante era hacer que estas consultas fueran **fáciles de entender** tanto para la persona que escribe la query, como para los que vienen después ya sea a modificarla como para hacer la review.

Lo que se consiguió fue un lenguaje muy sencillo de entender ya que utiliza extension methods que son declarados de una forma similar a las expresiones SQL.

Por ejemplo, la cláusula `.Where()` dentro de LINQ nos permite filtrar.

Para realizar esta acción, tenemos dos opciones, ambas nos otorgan una sintaxis entendible y sencilla y por supuesto comprobación de los tipos en tiempo de compilación.

Forma numero 1 de escribir consultas LINQ:

```
public static void LinqQueryForma1(Libro[] arrayLibros, string autor)
{
    var libros = from libro in arrayLibros
                 where libro.Autor == autor
                 select libro ;
}
```

Como vemos disponemos de un método que recibe un `array` y un `string` autor, el cual lo utilizaremos para filtrar la lista. Si sabes SQL puedes comprobar que LINQ utiliza una sintaxis muy similar.

Al final de la consulta, indicamos una cláusula `select libro` la cual nos va a devolver un tipo `IEnumerable<T>` donde `T` es `Libro`.

Si en vez de devolver libro, quisiéramos devolver solo el título, `T` sería un `string`.

Forma numero 2 de escribir consultas LINQ

La segunda forma es utilizando extension methods llamándolos uno detrás de otro.

Replicar el ejemplo anterior es muy sencillo, únicamente tenemos que indicar el siguiente código:

```
public static void LinQueryForma2(Libro[] arrayLibros, string autor)
{
    var libros = arrayLibros.Where(a => a.Autor == autor);
}
```

Como vemos `.Where` es un extension method, el cual acepta un delegado `Func` por parámetro. El cual filtrará cuando sea verdadero y falso, devolviendo la lista filtrada.

Como podemos observar ya no incluimos el `.Select` y es porque por defecto el código entiende que queremos hacer un select.

además de un `.Where` podemos concatenar más acciones, si por ejemplo queremos ordenar por el título podemos hacerlo de la siguiente manera:

```
public static void LinQueryForma2(Libro[] arrayLibros, string autor)
{
    var libros = arrayLibros.Where(a => a.Autor == autor).OrderBy(a=>a.Titulo);
}
```

Ambas opciones son igual de válidas, personalmente prefiero la segunda, pero ambas nos permiten realizar consultas que lucen como consultas reales y fáciles de entender.

2.3 La interfaz IEnumerable

Como acabamos de ver, cuando hacemos una consulta LINQ el resultado nos viene en un tipo `IEnumerable`.

Esta interfaz es la más importante cuando estamos utilizando LINQ ya que la gran mayoría (el 98%) de extension methods que vamos a utilizar lo realizan sobre `IEnumerable<T>`.

Como es un tipo genérico, nos permite indicar qué tipo vamos a introducir en la colección.

2.3.1 Qué hace por detrás

El motivo por el que podemos hacer un `foreach` sobre el resultado de la query o el motivo por el que podemos hacer un `foreach` sobre el array que hemos creado al principio, es debido a que ambos tipos implementan la interfaz `IEnumerable`, que tiene un método llamado `GetEnumerator()`.

El método `GetEnumerator()` de `IEnumerable` viene directamente de heredar `IEnumerator`. Y si convertimos cualquiera de nuestros elementos, en mi caso `arraylibros` en `IEnumerable<Libros>` puedo acceder a `arrayLibros.GetEnumerator()` y una vez tienes este enumerador, puedes pedir al enumerador que se mueva al siguiente elemento utilizando `.MoveNext()` y podremos acceder a el con la propiedad `.Current` que contiene un puntero al elemento que estamos iterando, con lo que podemos acceder a su valor.

```
IEnumerable<Libro> arrayLibros = new Libro[] {
    new Libro(1, "Poeta en NY", "Federico García Lorca"),
    new Libro(2, "Los asesinos", "S. Posteguillo"),
    new Libro(3, "circo máximo", "S. Posteguillo"),
    new Libro(4, "Frankenstein", "S. Posteguillo"),
    new Libro(5, "El origen perdido", "Matilde Asensi")
};
```

```
IEnumerator<Libro> enumeratorLibros = arrayLibros.GetEnumerator();
while (enumeratorLibros.MoveNext())
{
    Console.WriteLine(enumeratorLibros.Current);
}
```

Lo bueno de utilizar `IEnumerable` es que es una interfaz que puede englobar, un array, una lista, o una consulta a una base de datos, por lo que queda totalmente transparente al usuario lo que significa que es muy fácil de mantener y trabajar con ello.

Antes de pasar al siguiente punto, es importante remarcar que la interfaz `IEnumerable` es "Lazy" lo que significa que no va a ser ejecutada hasta que necesitamos su ejecución u obtener un elemento fuera del `IEnumerable`, por ejemplo, en un `foreach`, o si convertimos todo el `IEnumerable` en una `List<T>`.

2.4 Funcionalidades para construir queries en LINQ

Por defecto LINQ nos trae una gran variedad de métodos para construir nuestras consultas, y en el 99.9% de los casos, serán más que suficientes.

Las funcionalidades de las que disponemos para construir queries en LINQ son muy similares al SQL normal y así son sus nombres, no voy a entrar en detalle en todas ellas pero si en las principales.

La gran mayoría de extension methods en LINQ utilizan un delegado como parámetro de entrada que es `Func<T, bool>`

2.4.1 Where en LINQ

El principal y más importante es el método .Where el cual nos permite filtrar utilizando los parámetros de nuestra consulta. por ejemplo como el caso que hemos visto, Comparando el nombre.

```
var libros = arrayLibros.Where(a => a.Autor == autor);
```

Cuando utilizamos where nos devuelve una lista, esto quiere decir que tendrá 0 o más elementos.

2.4.2 Obtención de un único elemento con LINQ

Para obtener un único elemento tenemos varias opciones, TODAS ellas, reciben como parámetro el delegado `Func<T, bool>` (igual que el where) y devuelven un elemento del tipo `T`.

- `.First()` -> devuelve el primer elemento
- `.FristOrDefault()` - > devuelve el primer elemento o uno por defecto.
- `.Single()` -> si hay más de un elemento o no hay ninguno, devuelve una excepción, si hay un elemento lo devuelve.

- `.SingleOrDefault()` -> en caso de haber más de un elemento salta una excepción y si hay solo uno o ningún devuelve el elemento o uno por defecto.

¿que utilizar? `.Single`, o `.First`, la respuesta es que ninguno, ya que `.First` puede devolver falsos positivos si hay más de un elemento y `.Single` necesita leer toda la enumeración para comprobar que no hay ninguno más.

Depende un poco del escenario, pero para obtener elementos únicos que no sabemos a ciencia cierta que son un ID único en la base de datos recomiendo una combinación entre `.Count == 1` y `.First`. haciendo que si count no es 1 salte una excepción.

2.4.3 Último elemento con LINQ

Similar al anterior.

- `.Last()` -> devuelve el último elemento.

- `.LastOrDefault()` -> devuelve el último elemento o uno por defecto (vacío).

2.4.4 Ordenar elementos en LINQ

Para ordenar elementos debemos llamar al método `.OrderBy` u `.OrderByDescending` el cual ordenará de forma descendente.

Ambos métodos contienen un método adicional en el que puedes indicar un objeto del tipo `IComparer` para poder ordenar a tu gusto.

El resultado que obtendremos es la propia lista ordenada como hemos indicado:

```
var librosOrdenas = arrayLibros.Where(a => a.Autor == "S. Posteguillo").OrderBy(a =>
a.Titulo);
```

2.4.5 Agrupar elementos en LINQ

Para agrupar elementos debemos utilizar el método `.GroupBy()` y el resultado será una lista `IEnumerable<Grouping<Key, T>>` en nuestro caso, agrupamos por autor, y podemos iterar sobre la lista, accediendo al grupo, key contiene el nombre del autor del libro.

```
var agrupacion = arrayLibros.GroupBy(a => a.Autor);
```

```
foreach(var autorLibro in agrupacion)
{
    Console.WriteLine(autorLibro.Key);

    foreach(var libro in autorLibro)
    {
        Console.WriteLine(libro.Titulo);
    }
}
```