C# - Expresiones Lambda

Contenido

1 Expresiones Lambda 2
1.1 A modo de introducción 2
1.1.1 Sintaxis 2
1.1.2 Puntos importantes: 3
1.1.3 Los delegados genéricos: action y func 3
1.1.4 Lambdas de expresión 4
1.1.5 Lambdas de instrucciones 4
1.2 Expresiones Lambda y genéricos 5
1.2.1 Un Ejemplo 6
1.3 Lambdas de Async 7
1.4 Lambdas con los operadores de consulta estándar 8
1.4.1 Ejemplos de Expresiones Lambda en LINQ 8
1.5 Deducción de tipos en las expresiones lambda 8
1.6 Referencia Bibliográfica 9

1 Expresiones Lambda

1.1 A modo de introducción

Una expresión lambda es una expresión que devuelve un método

Una expresión lambda es una función anónima que puede contener expresiones e instrucciones y se puede utilizar para crear delegados o tipos de árboles de expresión.

Esto resulta bastante extraño porque la mayoría de las expresiones devuelven (más o menos) un valor

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.

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.

1.1.1 Sintaxis

```
( () => {Folder.StopFolding[0];} )
() => { statusBarItemInformes.Content = texto; }
(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 vacios
- 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 identar 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 alinea igual que en un método ordinario
- Las expresiones lambda no se permiten en el lado izquierdo del operador is o as.

- 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;
                };
```

- No es necesario especificar el tipo de los parámetros, ya que siempre van asociados a un delegado que ya contiene esta información.

```
delegate string Delegado(int edad);
[...]
Delegado edad = e => string.Format("Tu edad es {0}", e);
```

- Se pueden usar tanto por delegados creados por nosotros como por aquellos definidos por el framework.

```
<%= Html.TextBoxFor(m => m.Name) %>
```

1.1.3 Los delegados genéricos: action y func

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

1.1.4 Lambdas de expresión

Una expresión lambda con una expresión en el lado derecho se denomina lambda de expresión. Las lambdas de expresión se utilizan ampliamente en la construcción de Árboles de expresión (C# y Visual Basic).Una lambda de expresión devuelve el resultado de la expresión y presenta la siguiente forma básica:

```
(input parameters) => expression
```

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:

```
 (input parameters) => {statement;}
```

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.

```
delegate void TestDelegate(string s);
…
TestDelegate myDel = n => { string s = n + " " + "World"; Console.WriteLine(s); };
myDel("Hello");
```

Las lambdas de instrucciones, como los métodos anónimos, no se pueden utilizar para crear árboles de expresión.

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 –recuerde el término, que comenzará a utilizar con frecuencia en un futuro cercano- 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. En la versión original de C# habría sido necesario definir explícitamente la función (y el delegado no habría podido ser genérico):

```
static int Cuadrado(int x)
{
return x * x;
}
Mapeado<int> cuadrado2 = Cuadrado;
```

Aunque los métodos anónimos ofrecen una notación más compacta y directa, su sintaxis es aún bastante verbosa y de naturaleza imperativa. En particular, al definir un método anónimo es necesario:

- Utilizar explícitamente la palabra reservada delegate.
- Indicar explícitamente los tipos de los parámetros, sin ninguna posibilidad de que el compilador los deduzca a partir del contexto de utilización.
- En el caso bastante frecuente en que el cuerpo de la función es una simple sentencia return seguida de una expresión que evalúa el resultado (como en el caso de nuestro cuadrado), se hace aún más evidente el exceso sintáctico.

Las expresiones lambda pueden considerarse como una extensión de los métodos anónimos, que ofrecen una sintaxis más concisa y funcional para expresarlos. La sintaxis de una expresión lambda consta de una lista de variables-parámetros, seguida del símbolo de implicación (aplicación de función) =>, que es seguido a su vez de la expresión o bloque de sentencias que implementa la funcionalidad deseada. Por ejemplo, nuestra definición de cuadrado quedaría de la siguiente forma utilizando una expresión lambda:

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

¿Más corto imposible, verdad? 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;
```

Si se especifica el tipo del parámetro, o la expresión tiene más de un parámetro (se indiquen explícitamente o no sus tipos) los paréntesis son obligatorios. Por otra parte, en la parte derecha de la implicación se puede colocar una expresión, como hemos hecho hasta el momento, o un bloque de sentencias de cualquier complejidad (siempre que por supuesto produzca como resultado un valor del tipo adecuado):

```
Mapeado<int> cuadrado5 = (int x) => { return x * x; };
```

Aunque para una mejor comprensión hemos definido nuestro propio tipo delegado genérico Mapeado<T>, en la práctica la mayor parte de las veces utilizaremos las distintas sobrecargas del tipo genérico predefinido Func:

```
// en System.Query.dll
// espacio de nombres System.Query
public delegate T Func<T>();
public delegate T Func<A0, T>(A0 a0);
public delegate T Func<A0, A1, T>(A0 a0, A1 a1);
public delegate T Func<A0, A1, A2, T>(A0 a0, A1 a1, A2 a2);
public delegate T Func<A0, A1, A2, A3, T>(A0 a0, A1 a1, A2 a2, A3 a3);
```

El tipo Func representa a los delegados a funciones (con 0, 1, 2 ó 3 argumentos, respectivamente) que devuelven un valor de tipo T. Utilizando este tipo, podríamos definir una nueva versión de cuadrado así:

```
Func<int, int> cuadrado6 = 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>).

Bien, pues básicamente las expresiones lambda son una sintaxis alternativa (mucho más compacta), para hacer exactamente lo mismo. Fijaros como quedaría el código anterior usando expresiones lambda:

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

Es una sintaxis mucho más compacta que "la clásica" de métodos anónimos. Vamos a comentarla rápidamente...

El operador => es el operador lambda, y la sintaxis es param-list => valor_retorno

- siendo param-list una lista de parámetros (separados por comas), y
- valor_retorno una expresión que se evaluará y será el resultado de la expresión lambda.

Veamos otro ejemplo de cómo las expresiones lambda sustituyen (sintácticamente) a un método anónimo. Imaginemos el delegate:

```
public delegate T Updater<T> (T value);
```

Un delegate que toma un objeto de tipo T y devuelve otro del mismo tipo.

En la sintaxis estándard (C# 1.0 sin métodos anónimos) haríamos algo como:

```
Updater<int> pFoo = new Updater<int>(SumaUno);
```

Donde SumaUno, sería un método definido como algo parecido a:

```
int SumaUno(int x) { return x + 1; }
```

Usando métodos anónimos (C# 2.0) la cosa nos queda como:

```
Updater<int> pFooAnonimo = delegate(int x) { return x + 1; };
```

Lo cual ya es más compacto que la sintaxis anterior... y usando las nuevas expresiones lambda, la cosa nos queda como:

```
Updater<int> pFooLambda = x => x + 1;
```

Bueno... una cosilla debéis tener presente con las expresiones lambda... C# es un lenguaje fuertemente tipado, y por ello las expresiones lambda tienen tipo... Que nosotros no pongamos tipo en los parámetros de una expresión lambda, no significa que lo tengan, significa que el compilador debe poder deducirlos por el contexto. En este ejemplo que acabo de poner, el compilador deduce que "x" es de tipo int, porque estamos asignando la expresión lambda a un delegate que espera un parámetro de tipo int (**y una expresión lambda siempre se asigna a un delegate, y es este delegate quien nos define el tipo de los parámetros**). En el otro ejemplo que he expuesto antes (el de la llamada a Sort), el compilador puede deducir que los parámetros x e y son de tipo string, porque estoy llamando a Sort de una List<string> que por lo tanto me espera un delegate de tipo Comparison<string> que define dos parámetros de tipo string.

1.3 Lambdas de Async

Más información en - Expresiones lambda (Guía de programación de C#)
http://msdn.microsoft.com/es-es/library/bb397687.aspx

1.4 Lambdas con los operadores de consulta estándar

Más información en – Expresiones lambda (Guía de programación de C#)
http://msdn.microsoft.com/es-es/library/bb397687.aspx

1.4.1 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 };
var numMayores = nums.Where(n => n > 5).ToList();
var numPares = nums.Where(n => n % 2 == 0).ToList();
var numImpares = nums.Where(n => n % 2 != 0).ToList();
```

El en código de ejemplo anterior tenemos 3 consultas LINQ:

- La primera consulta la usamos para guardar en la variable de tipo anónimo (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).

1.5 Deducción de tipos en las expresiones lambda

Al escribir expresiones lambda, no tiene por qué especificar un tipo para los parámetros de entrada, ya que el compilador puede deducir el tipo según el cuerpo de la lambda, el tipo de delegado subyacente y otros factores que se describen en la especificación del lenguaje C#.Para la mayoría de los operadores de consulta estándar, la primera entrada es el tipo de los elementos en la secuencia de origen.Así, si está realizando una consulta sobre un IEnumerable<Customer>, se deducirá que la variable de entrada será un objeto Customer, lo cual significa que se podrá tener acceso a sus métodos y propiedades:

```
customers.Where(c => c.City == "London");
```

Las reglas generales para las expresiones lambda son las siguientes:

- La lambda debe contener el mismo número de parámetros que el tipo delegado.
- Cada parámetro de entrada en la lambda se debe poder convertir implícitamente a su parámetro de delegado correspondiente.
- El valor devuelto de la lambda (si existe) se debe poder convertir implícitamente al tipo de valor devuelto del delegado.

Observe que las expresiones lambda, en sí mismas, no tienen tipo, ya que el sistema de tipos comunes no tiene ningún concepto intrínseco de "expresión lambda". Sin embargo, a veces resulta práctico hablar informalmente del "tipo" de una expresión lambda. En estos casos, el tipo hace referencia al tipo del delegado o el tipo de Expression en el que se convierte la expresión lambda.

Más información en - Expresiones lambda (Guía de programación de C#)
http://msdn.microsoft.com/es-es/library/bb397687.aspx

1.6 Referencia Bibliográfica

- Expresiones lambda (Guía de programación de C#)
  - http://msdn.microsoft.com/es-es/library/bb397687.aspx
- Árboles de expresión (C# y Visual Basic)
  - http://msdn.microsoft.com/es-es/library/bb397951.aspx
- Las expresiones lambda en C# 3.0
  - http://www.elguille.info/NET/futuro/firmas_octavio_ExpresionesLambda.htm
- Comparison<T> Delegate
  - http://msdn.microsoft.com/en-us/library/tfakywbh.aspx
- C# es fácil (iv): Expresiones lambda
  - http://burbujasnet.blogspot.com.es/2008/06/c-es-fcil-iv-expresiones-lambda.html
- Func<T, TResult> (Delegado)
  - http://msdn.microsoft.com/es-es/library/bb549151.aspx
- Action (Delegado)
  - http://msdn.microsoft.com/es-es/library/system.action.aspx
  - http://msdn.microsoft.com/es-es/library/bb548654.aspx
- Expresiones lambda
  - http://geeks.ms/blogs/gtorres/archive/2010/02/22/expresiones-lambda.aspx
- Cálculo lambda (wikipedia)
  - http://es.wikipedia.org/wiki/C%C3%A1lculo_lambda

Información sobre el documento
- **dc.date.created:** 2012-08-28 (Fecha Creación)
- **dc.date.modified:** 2012-09-03 (Fecha Modificación)
- **dc.date.available:** 2012-09-03 (Fecha Impresión)
- **dc.author:** Joaquin Medina Serrano [joaquin@medina.name]