

Entendiendo las expresiones lambda (lambda expressions) en C#

por Martín Márquez <xomalli@gmail.com>

Introducción

La programación imperativa es uno de los paradigmas de computación mas ampliamente utilizados por la mayoría de lenguajes de programación de alto nivel debido al gran soporte académico y comercial y a que los programas son relativamente independientes del tipo de computadora donde se ejecutan porque los lenguajes de programación deben abstraer las operaciones del modelo de maquina para la cual se diseñaron.

La programación imperativa se basa en el modelo de la máquina de Von Neuman, del cual la mayoría de las computadoras personales y estaciones de trabajo tienen elementos comunes.

Aunque menos utilizado existe otro paradigma que a diferencia del imperativo se basa en las matemáticas (aplicación de funciones) con el cual igualmente podemos expresar operaciones computacionales de forma mas compacta y abstracta, este paradigma se conoce como programación funcional.

Uno de los muchos elementos del paradigma funcional que .NET incluye desde la versión 3.0 son las expresiones lambda (lambda expression).

Programación Funcional

Los conceptos básicos de la programación funcional datan de 1956 a 1958 con el trabajo de Jonh McCarthy en el desarrollo y diseño de LISP (List Processor), este lenguaje esta basado en el calculo lambda que sento las bases de los lenguajes funcionales, características como:

- **Recursion:** se utiliza para para realizar operaciones repetitivas, no utiliza la iteración.
- **Funciones de primer orden:** las funciones tienen el mismo nivel que cualquier otro elemento del lenguaje, pueden aplicarse a valores, evaluarse, regresar un valor y ser parámetros de otras funciones.
- **No requiere asignación:** el computó se realiza aplicando funciones a los argumentos.
- **Garbage collector:** Se reclaman los objetos que no están siendo utilizado por el programa.
- **Tipado dinámico (Dynamic typing):** La comprobación del tipo se realiza en tiempo de ejecución, los valores tienen tipos pero las variables no.

El paradigma funcional se basa en el concepto matemático de función, que la mayoría de los lenguajes de programación imperativos y funcionales comparten y cuya definición es

(1) Una función f es una regla que asigna a cada elemento x de un conjunto A exactamente un elemento llamado $f(x)$ de conjunto B

Donde la programación funcional marca su diferencia con la imperativa es que para la programación funcional cada programa es equivalente a esta definición donde x es el argumento o dominio de f mientras que y es el rango de f o la salida sea los programas son cajas negras donde solo importa el que se esta computando y no el como se esta computando que es el caso de la programación imperativa.

En resumen cuando se programa de forma funcional se piensa más en expresiones y su significado que en una secuencia de operaciones en memoria.

Tipos Delegate y métodos anónimos

Desde sus primeras versiones .NET introdujo el objeto **delegate (delegado)** que es un tipo particular de objeto (un delegate deriva de la clase base System.Delegate), que puede encapsular la referencia a un método estático o de una instancia como si fuera un mecanismo de callback(devolución de llamada) similar a los apuntadores de función de C y C++ pero con la importante diferencia de que proporciona un tipado seguro (type-safety) para evitar errores en tiempo de ejecución y que puedan detectarse en tiempo de compilación si la función no coincide con la firma del método al que hace referencia.

Esto posibilita en un contexto de programación orientada a objetos que los métodos pueden recibir como argumentos otros métodos además de tipos primitivos y de referencia. Veamos un ejemplo para ilustrar estos conceptos con C#.

En versiones anteriores a C# 2.0 (1.1,1.0) los delegate se utilizaban como parámetros, veamos el siguiente ejemplo que se compone de dos clases:

1. La clase **Temp.cs** que es la clase que encapsula las operaciones.
2. La clase **Program.cs** que es la clase principal que asigna los métodos al delegado.

Ejemplo 1.1 Uso de métodos como parámetros en C# 1.1.

Aquí código fuente completo de la clase **Temp.cs**

```
using System;
namespace Samples.Delegates
{
    public class Temp
    {
        //la implementación de cada método
        public static double GetCelsius(double fahrenheit)
        {
            return (fahrenheit - 32) * (5 / 9D);
        }

        public static double GetKelvin(double fahrenheit)
        {
            return fahrenheit + 460;
        }
    }
}
```

Aquí código fuente completo de la clase **Program.cs**

```
using System;

namespace Samples.Delegates
{
    class MainClass
    {
        //definimos al objeto que guardara las referencias a los métodos
        delegate double GetTemp(double d);
        public static void Main(string[] args)
        {
            int x = 44;
            Console.WriteLine("{0} Fahrenheit = {1:0.00} Celsius", x,
ApplyF(x, Temp.GetCelsius));
            Console.WriteLine("{0} Fahrenheit = {1:0.00} Kelvin", x,
ApplyF(x, Temp.GetKelvin));
            Console.Read();
        }

        //el método que aplicará el método que es su segundo argumento
        static double ApplyF(double d, GetTemp f)
        {
            return f(d);
        }
    }
}
```

Observamos que los métodos que implementan la funcionalidad deben declararse de una manera completamente procedural e imperativa.

```
public static double GetCelsius(double fahrenheit)
{
    return (fahrenheit - 32) * (5 / 9D);
}

public static double GetKelvin(double fahrenheit)
{
    return fahrenheit + 460;
}
```

Las versiones posteriores de C# al incorporar los métodos anónimos se acerca más a la programación funcional al asociar un bloque de código a un *delegate* sin necesidad de tener toda su implementación en un método sino dentro de la misma declaración del objeto, como se muestra en el siguiente código que es el mismo código anterior pero usando métodos anónimos.

Ejemplo 1.2 Uso de métodos como parámetros utilizando métodos anónimos.

```
using System;

namespace Samples.AnonymousMethods
{
    class MainClass
    {
        delegate double GetTemp(double d);
```

```

public static void Main(string[] args)
{
    int x = 44;
    Console.WriteLine("{0} Fahrenheit = {1:0.00} Celsius", x,
    ApplyF(x, delegate (double fahrenheit)
    {
        return (fahrenheit - 32) * (5 / 9D);
    }));
    Console.WriteLine("{0} Fahrenheit = {1:0.00} Kelvin", x,
    ApplyF(x, delegate (double fahrenheit)
    {
        return fahrenheit + 460;
    }));
    Console.Read();
}

static double ApplyF(double d, GetTemp f)
{
    return f(d);
}
}

```

Aquí observamos la diferencia con respecto al código anterior del Ejemplo 1.1.

```

    Console.WriteLine("{0} Fahrenheit = {1:0.00} Celsius", x, ApplyF(x,
    delegate (double fahrenheit)
    {
        return (fahrenheit - 32) * (5 / 9D);
    }));
    Console.WriteLine("{0} Fahrenheit = {1:0.00} Kelvin", x,
    ApplyF(x,
        delegate (double fahrenheit)
        {
            return fahrenheit + 460;
        }));

```

Expresiones Lambda (Lambda Expressions)

Las expresiones lambda provienen del cálculo lambda (lambda calculus) desarrollado por Alonzo Church en los años 1930's como una notación para representar todas las funciones computables equivalentes a una máquina de Turing, todos los lenguajes funcionales pueden ser vistos como una variante sintáctica del cálculo lambda.

Las expresiones Lambda son útiles para sintetizar funciones con pocos parámetros que regresan algún valor, esta expresión consiste básicamente en una regla de sustitución que expresa tal cual una función o sea un mapeo de los elementos del conjunto dominio a los elementos de un codominio por ejemplo en la siguiente expresión:

cuadrado : integer \rightarrow integer donde cuadrado(n) = n^2

Se reduce a una notación que produce una función anónima donde los únicos símbolos son la letra **lambda** (λ) y el punto (.).

$\lambda n. n^2$

En la mayoría de los lenguajes funcionales las funciones anónimas son valores representados por la palabra reservada **lambda**, como el caso de LISP.

Lambda (n)(**n)

Aunque C# no utiliza los símbolos de la notación matemática lambda, el operador lambda es => que significa “tiende a” o “va hacia a”, la estructura de una expresión lambda en C# es:

(Argumentos de entrada) => (salida al procesarlos)

En caso de únicamente una variable la sintaxis es:

(x) => (x*x)

En caso de múltiples argumentos la sintaxis es:

(x,y,z) => ()

Aquí es importante saber que es el tipo delegate que dicta el tipo de los parámetros de entrada y de salida.

Ejemplo 1.3 Métodos como parámetros utilizando expresiones lambda.

```
using System;

namespace Samples.LambdaExpression
{
    class MainClass
    {
        delegate double GetTemp(double d);
        public static void Main(string[] args)
        {
            int x = 44;
            Console.WriteLine("{0} Fahrenheit = {1:0.00} Celsius", x,
                ApplyF(x, (fahrenheit) => ((fahrenheit - 32) * (5 / 9D))));
            Console.WriteLine("{0} Fahrenheit = {1:0.00} Kelvin", x,
                ApplyF(x, (fahrenheit) => (fahrenheit + 460)));
            Console.Read();
        }

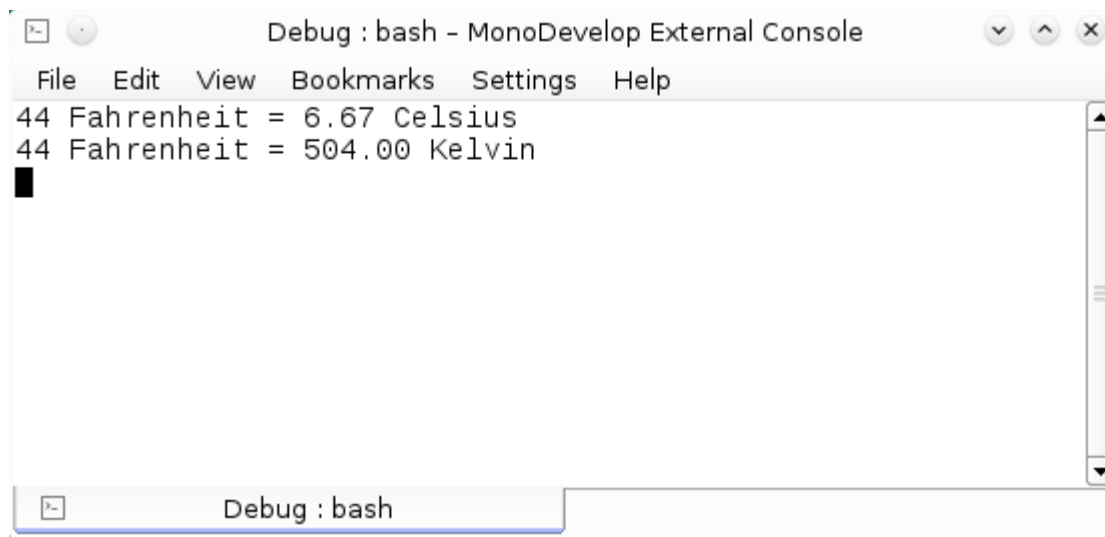
        static double ApplyF(double d, GetTemp f)
        {
            return f(d);
        }
    }
}
```

Observamos que de los métodos anónimos a las expresiones Lambda, nos queda una sintaxis más legible y compacta.

```
ApplyF(x, (fahrenheit) => ((fahrenheit - 32) * (5 / 9D))); ApplyF(x, (fahrenheit) => (fahrenheit + 460));
```

Al ejecutar todos los programas observamos que el resultado es el mismo en cada uno, lo que cambio fue únicamente la sintaxis.

Fig 1. La salida es identica por cada uno de los programas.



Conclusión

Para los nuevos retos en el desarrollo de software, es importante que los lenguajes de programación incorporen características de un paradigma de programación diferente a ellos para extender sus capacidades y así poder expresar algoritmos de una manera compacta y más concisa esto da como resultado un código más legible.



[Download el código fuente del ejemplo 1.1 *Samples.Delegates* para Xamarin o VS](#)



[Download el código fuente del ejemplo 1.2 *Samples.AnonymousMethods* para Xamarin o VS](#)



[Download el código fuente del ejemplo 1.3 *Samples.LambdaExpression* para Xamarin o VS](#)

Este documento está protegido bajo la licencia de documentación libre *Free Documentacion License* del Proyecto GNU, para consulta ver el sitio <http://www.gnu.org/licenses/fdl.txt> , toda persona que lo desee está autorizada a usar, copiar y modificar este documento según los puntos establecidos en la «Licencia FDL»