PREGRADO

UNIDAD 1 | OVERVIEW

C# FUNCTIONAL FEATURES

SI730 | Aplicaciones Web

Al finalizar la unidad de aprendizaje, el estudiante describe el proceso de software realizado aplicando el paradigma orientado a objetos combinado con aspectos de la programación funcional, utilizando el lenguaje C# y frameworks de actualidad, para desarrollar aplicaciones web básicas en un ambiente de desarrollo colaborativo.

AGENDA

INTRO

DELEGATES

LAMBDA EXPRESSIONS

Intro

"Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style." ... "it is often possible to write functional-style programs in an imperative language, and vice versa. It is then a matter of opinion whether a particular language can be described as functional or not."

Intro

Lenguajes como F# cuentan con una gran cantidad de características funcionales, sin embargo, el lenguaje C# que podría llamarse lenguaje imperativo, cuenta con varias características funcionales.

Immutability

.NET BCL ofrece immutable collections, value types y strings.

```
public class User
{
    public string FirstName { get; }
    public string LastName { get; }

    public User(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }


    public User WithFirstName(string firstName) => new User(firstName, LastName);
    public User WithLastName(string lastName) => new User(FirstName, lastName);
}


var user = new User("Jane", "Doe"); // Jane Doe
user = user.WithFirstName("Janet"); // Janet Doe
```

Expressions

Es posible usar expressions en vez de statements en varios casos.

```
int Add(int x, int y) => x + y;
int z = Add(1, 2);


bool equals1 = z == 3; // -> true
bool equals2 = z == z; // -> true
bool equals3 = z == Add(1, 2); // -> true
bool equals4 = Add(1, 2) == z; // -> true
```

Functions as Values

Functions son first-class citizens en C#. Es posible pasar static functions, class members o local functions.

```
string FirstName(User u) => u.FirstName;
string FullName(User u) => $"{u.FirstName} {u.LastName}";
string SayHello(Func<User, string> formatter, User u)
    => $"Hello, {formatter(u)}!";


var user = new User("Jane", "Doe");
SayHello(FirstName, user); // Hello, Jane!
SayHello(FullName, user); // Hello, Jane Doe!
```

AGENDA

INTRO

DELEGATES

LAMBDA EXPRESSIONS

Delegates

Un delegate es una referencia a un método.

Puede usarse un delegate object para pasar el código en el que queremos llamar a un reference method, sin saber en tiempo de compilación cuál método será invocado.

Tres pasos:

- Declarar el delegate
- Instanciar, crear el delegate object
- Invocar, donde llamamos al reference method

Delegates

Ejemplo

```
//Declaration
public delegate void WriterDelegate(string text);
class Program
{
    public static void Write(string text)
    {
        Console.WriteLine(text);
    }

    static void Main(string[] args)
    {
        //Instantiation
        WriterDelegate writerDelegate = new WriterDelegate(Write);

        //Invocation
        writerDelegate("Some example text.");
    }
}
```

Anonymous Method

Ejemplo

```
delegate void NumberChanger(int n);

namespace DelegateAppl {

   class TestDelegate {
      static int num = 10;

      public static void AddNum(int p) {
         num += p;
         Console.WriteLine("Named Method: {0}", num);
      }
      public static void MultNum(int q) {
         num *= q;
         Console.WriteLine("Named Method: {0}", num);
      }
      public static int getNum() {
         return num;
      }
//...
```

Anonymous Method

```
//...
static void Main(string[] args) {
   //create delegate instances using anonymous method
   NumberChanger nc = delegate(int x) {
      Console.WriteLine("Anonymous Method: {0}", x);
   };

   //calling the delegate using the anonymous method
   nc(10);

   //instantiating the delegate using the named methods
   nc =  new NumberChanger(AddNum);

   //calling the delegate using the named methods
   nc(5);

   //instantiating the delegate using another named methods
   nc =  new NumberChanger(MultNum);

   //calling the delegate using the named methods
   nc(2);
   Console.ReadKey();
      }
   }
}
```

Delegates

Dos built-in delegates

- Func<T>
- Action<T>

Func<T> Delegate

Encapsula un método que tenga hasta 16 parámetros y retorna un valor del tipo especificado (no void).

```
class Program
{
    public static int Sum(int a, int b)
    {
        return a + b;
    }

    static void Main(string[] args)
    {
        Func<int, int, int> sumDelegate = Sum;
        Console.WriteLine(sumDelegate(10, 20));
    }
}
```

Action<T> Delegate

Encapsula un método que tenga hasta 16 parámetros y no retorna ningún valor. Solo puede recibir métodos que retornen void.

```
public static void Write(string text)
{
    Console.WriteLine(text);
}

static void Main(string[] args)
{
    Action<string> writeDelegate = Write;
    writeDelegate("String parameter to write.");
}
```

Example

Operation manager con delegates.

```
public enum Operation
{
    Sum,
    Subtract,
    Multiply
}
```

Example

```
public class ExecutionManager
{
    public Dictionary<Operation, Func<int>> FuncExecute { get; set; }
    private Func<int> _sum;
    private Func<int> _subtract;
    private Func<int> _multiply;

    public ExecutionManager()
    {
        FuncExecute = new Dictionary<Operation, Func<int>>(3);
    }

    public void PopulateFunctions(Func<int> Sum, Func<int> Subtract, Func<int> Multiply)
    {
        _sum = Sum;
        _subtract = Subtract;
        _multiply = Multiply;
    }

    public void PrepareExecution()
    {
        FuncExecute.Add(Operation.Sum, _sum);
        FuncExecute.Add(Operation.Subtract, _subtract);
        FuncExecute.Add(Operation.Multiply, _multiply);
    }
}
```

Example

```
public class OperationManager
{
    private int _first;
    private int _second;
    private readonly ExecutionManager _executionManager;

    public OperationManager(int first, int second, ExecutionManager executionManager)
    {
        _first = first;
        _second = second;
        _executionManager = executionManager;
        _executionManager.PopulateFunctions(Sum, Subtract, Multiply);
        _executionManager.PrepareExecution();
    }

    private int Sum()
    {
        return _first + _second;
    }

    private int Subtract()
    {
        return _first - _second;
    }

    private int Multiply()
    {
        return _first * _second;
    }

    public int Execute(Operation operation)
    {
        return _executionManager.FuncExecute.ContainsKey(operation) ?_executionManager.FuncExecute[operation]() : -1;
    }
}
```

Example

```
class Program
{
    static void Main(string[] args)
    {
        var executionManager = new ExecutionManager();
        var operationManager = new OperationManager(20, 10, executionManager);
        var result = operationManager.Execute(Operation.Sum);
        Console.WriteLine($"The result of the operation is {result}");
        Console.ReadKey();
    }
}
```

AGENDA

INTRO

DELEGATES

LAMBDA EXPRESSIONS

Anatomy of Lambda expression

En esencia es un método que no tiene declaración, no tiene access modifier ni nombre.

Puede ser:

Expression lambda

Statement lambda

```
(Input parameters) => Expression or statement block
```

Lambda expressions

Un método anónimo como:

```
delegate(Author a) { return a.IsActive && a.NoOfBooksAuthored > 10; }
```

Puede convertirse en lambda expression:

```
(a) => { a.IsActive && a.NoOfBooksAuthored > 10; }
```

Lambda expressions

Otro ejemplo

```
List<int> integers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach(int num in integers.Where(n => n % 2 == 1).ToList())
{
  Console.WriteLine(num);
}
```

Expression lambda

Lambda expression sin parámetros

```
() => Console.WriteLine("This is a lambda expression without any parameter");
```

Expression lambda

Lambda expression con parámetros

```
(a, numberOfBooksAuthored) => a.NoOfBooksAuthored >= numberOfBooksAuthored;
```

Expression lambdas

Puede especificar el tipo de un parámetro

```
(a, int numberOfBooksAuthored) => a.NoOfBooksAuthored >= numberOfBooksAuthored;
```

Statement lambdas in C#

Idénticas a expression lambdas, pero en vez de tener una expresión a la derecha, tiene un bloque de código con con varias sentencias.

```
(a, 10) =>
{
    Console.WriteLine("This is an example of a lambda expression
                       with multiple statements");
    return a.NoOfBooksAuthored >= 10;
}
```

Statement lambdas in C#

Puede incluir sentencias de control de flujo.

```
int[] integers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int i in integers.Where(x =>
{
    if (x % 2 == 0)
        return true;
    return false;
}
))
Console.WriteLine(i);
```

RESUMEN

Recordemos

C# siendo principalmente un lenguaje imperativo, ofrece soporte para varias características de programación funcional.

Los delegates permiten trabajar con funciones anónimas, Func<T> y Action<T>.

C# brinda soporte para lambda expressions, incluyendo expression lambdas y statement lambdas.

REFERENCIAS

Para profundizar

Functional Programming FAQ
http://www.cs.nott.ac.uk/%7Epszgmh/faq.html

Delegates (C# Programming Guide)
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/

Lambda Expressions (C# Programming Guide)
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

PREGRADO

Ingeniería de Software

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova