Lambda es una forma reducida de crear funciones anónimas.

Hay 3 tipos de delegados implícitos en C#:

Predicate, Action y Func.

Los delegados son expresiones lambdas, anónimas.

Función anónima porque no tiene nombre.

Predicate siempre devuelve un booleano.

Una expresión lambda se la puede representar con la palabra reservada Func. Se le manda un tipo genérico 2. Si es una solo es la salida, si se mandan 2, habrá entrada y salida.

Action, otra palabra reservada, es como un void.

Dice que una de las ventajas que tiene es que se pueden llamar métodos y variables que están por fuera del Lambda.

Func<int, int> b = (a) => a \* 2;

Console.WriteLine( b(4));

/\* b es el nombre de la variable. El 4 es el parámetro que se pasa a la función.

\* El primer int representa el input, el segundo el output

\* Dice HDeLeon que b es un delegado

\* El return está implícito

\*/

Func<int, int, int> multiplica = (a, c) => a \* c;

Console.WriteLine(multiplica(4, 5));

//Ahí le paso 2 parámetros.

Console.WriteLine("------------------------------------"+Environment.NewLine);

//Sentencias lambda. Con mas de una linea, llaves y un return explícito.

Func<int, int, int> mayor = (a, c) =>

{

if (a > c) return a;

else if (a < c) return c;

else return 0;

};

int result = mayor(4, 10);

Console.WriteLine(result);

Console.WriteLine("------------------------------------" + Environment.NewLine);

//Ahora algo mas jodido:

List<int> numeros = new List<int>() { 3,6,7,4,1,10,9,8};

Func<int, bool> ObtenPares = (numero) => numero % 2 == 0;

var pares = numeros.Where(ObtenPares);

foreach (var item in pares)

{

Console.Write(item + " . ");

}

Action<int> Printeo = (numero) => Console.WriteLine("\n----------------------------------------\n" + numero);

Printeo(5);

Printeo(12);

Console.WriteLine("\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_-\_\n");

Action<List<int>> PrinteaLista = (lista) =>

{

lista.ForEach(item => Console.Write(item+" - "));

};

PrinteaLista(numeros);

Console.WriteLine("\n-------------------------------------\n");

//Funcion que recibe por parámetro otra funcion (Oooohhhh!!!)

Func<int, Func<int, int>, int> FuncionSuperior = (numero, FuncionInferior) =>

{

if (numero > 100)

{

return FuncionInferior(numero);

}

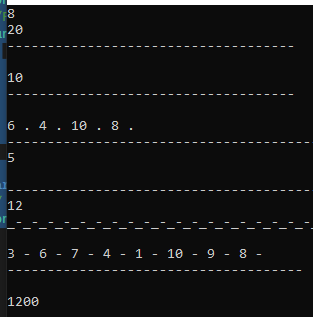
else return numero;

};

var res = FuncionSuperior(600, n => n \* 2);

// n=>n\*2 es la segunda funcion, la de orden inferior.

Console.WriteLine(res);



Lambdas reemplazando ifs

class Program

{

static void Main(string[] args)

{

var data = new Beer() { Name = "Minerva", Country = "México" };

Console.WriteLine(BeerValidations.NotNull(data.Name));

Console.ReadKey();

}

}

public class BeerValidations

{

public static readonly Predicate<string> NotNull = (d) => d != null;

}

public class Beer

{

public string Name { get; set; }

public string Country { get; set; }

}

Eso lo que hace es mandarle a Beervalidations.NotNull un string “Minerva”, como no está vacío, imprimirá true.

class Program

{

static void Main(string[] args)

{

var beer = new Beer() { Name = "Minerva", Country = "México" };

Console.WriteLine(BeerValidations.NotNullName(beer));

Console.WriteLine(BeerValidations.NotNullCountry(beer));

Console.ReadKey();

}

}

public class BeerValidations

{

public static readonly Predicate<Beer> NotNullName = (d) => d.Name != null;

public static readonly Predicate<Beer> NotNullCountry = (d) => d.Country != null;

}

public class Beer

{

public string Name { get; set; }

public string Country { get; set; }

}

Acá es lo mismo, pero al predicate se le pasa por parámetro todo el objeto.

class Program

{

static void Main(string[] args)

{

var beer = new Beer() { Name = "Minerva", Country = "México" };

Console.WriteLine(validator.Validate(beer, BeerValidations.validations));

Console.ReadKey();

}

}

public class BeerValidations

{

public static readonly Predicate<Beer>[] validations =

{

(d) => d.Name != null,

(d) => d.Country != null,

(d) =>d.Name.Count()<10,

(d)=> d.Country.Count()<5,

};

}

public class validator

{

public static bool Validate(Beer beer, params Predicate<Beer>[] validations) =>

validations.ToList().Where(d =>

{

return !d(beer);

}).Count() == 0;

}

public class Beer

{

public string Name { get; set; }

public string Country { get; set; }

}

Acá se enquilombó. El predicate beer recibe un array ahora.

Va a llevar a cabo las 4 validaciones que están en su cuerpo. Si todas son correctas devuelve true Count()==0 sería efectivamente 0.

NicosionRed

Relaciona a los delegados con los eventos, y puntuamente con sus métodos handler:

Un "Handler" es un método controlador de eventos en el receptor de eventos, es decir, el que se ejecuta en el momento de que el receptor es notificado de que ha ocurrido un evento. Para esto, se debe relacionar el método con el evento que va a controlar cuando ocurra.

Func

Sirve para crear delegados, que a diferencia de action, no son de tipo void.