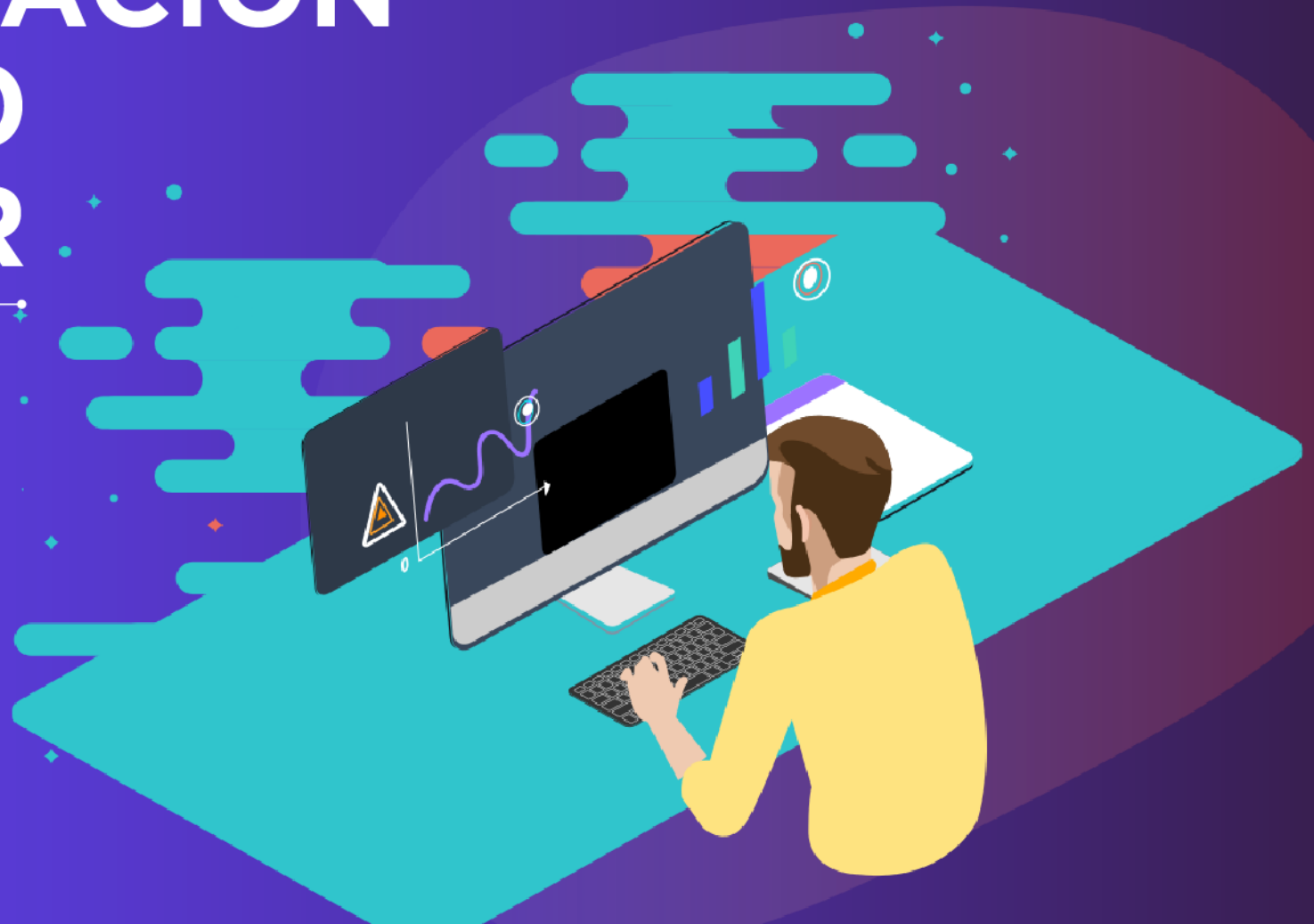
PROGRAMA

.NET ESPECIALIZACIÓN: NET

ESPECIALIZACIÓN ASP.NET 5.0 DEVELOPER



CURSO VIRTUAL

Fundamentos de Programación

Programación funcional

Instructor: Erick Aróstegui
earostegui@galaxy.edu.pe

GALAXY TRAINING

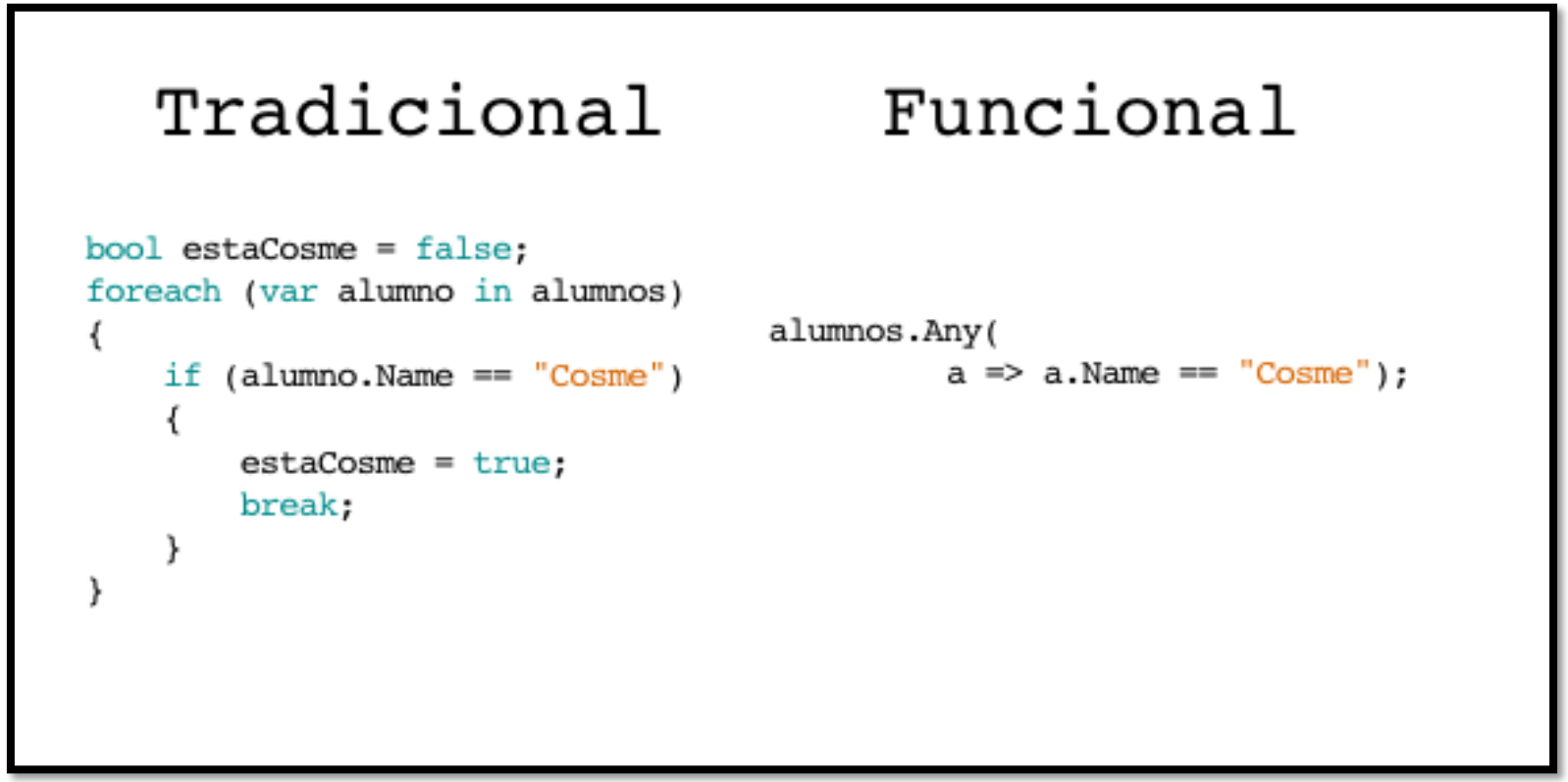
TEMAS

01	Principales conceptos y ventajas
02	Uso de Lambda Expressions
03	Delegados(Func y Action), inmutabilidad y colecciones
04	Funciones de orden superior, tuplas, cierres y recurrencia
05	Introducción a funciones asíncronas y funciones parciales

Principales conceptos

La programación funcional es un paradigma de programación, es decir, es de resolver diferentes problemáticas .Las funciones podrán ser asignadas a variables además podrán ser utilizadas como entrada y salida de otras funciones. A las funciones que puedan tomar funciones como parámetros y devolver funciones como resultado serán conocidas como función de orden superior.

La programación funcional es un paradigma declarativo, significa que nos enfocaremos en "qué" estamos haciendo y no en "cómo" se está haciendo que sería el enfoque imperativo. Esto quiere decir que nosotros expresaremos nuestra lógica sin describir controles de flujo; no usaremos ciclos o condicionales



Compara estos dos bloques, ambos en C#, en ambos estamos buscando valores dentro de una colección. Mientras que del modo tradicional le estamos diciendo al lenguaje que tiene que recorrer el arreglo y buscar el valor, del otro solamente le estamos diciendo que tiene que buscarlo.

Ventajas

- Más fáciles de escribir, depurar y mantener que los lenguajes imperativos gracias a la ausencia de efectos de borde
- El código se puede testear fácilmente
- Código mas preciso y más corto
- Fácil de combinar con la programación imperativa y orientada a objetos
- Muy adecuados para la paralelización

Lambda Expression

Es una función anónima que puede contener expresiones e instrucciones y se puede utilizar para crear delegados o tipos de árboles de expresión. Las expresiones lambda está formada por tres elementos

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

Sintaxis

- () 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.
- Las expresiones lambda no se permiten en el lado izquierdo del operador is o as
- Podemos pasar parámetros por referencia mediante la palabra clave [ref]

```
1  using System;
2
3  namespace Samples
4  {
5      public class Lambda
6      {
7          delegate int a(int b, int c);
8          public static int Main(string[] args)
9          {
10             Print(5, 5, (a, b) => a + b);
11             Print(4, 2, (x, y) => x * y);
12             Print(6, 6, (a, b) => a + b);
13             Print(144, 666, (x, y) => x * y);
14             return 0;
15         }
16
17         static void Print(int a, int b, a del)
18         {
19             Console.WriteLine("Imprimiendo: {0}", del(a, b));
20         }
21     }
22 }
```

Puntos importantes

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

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

Delegados

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto.

Los delegados se utilizan para pasar métodos como argumentos a otros métodos. Los controladores de eventos no son más que métodos que se invocan a través de delegados. Cree un método personalizado y una clase, como un control de Windows, podrá llamar al método cuando se produzca un determinado evento.

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

Delegados(Func)

Func proporciona un soporte para funciones anónimas parametrizadas. Los tipos principales son las entradas y el último tipo es siempre el valor de retorno.

```
1
2   // Declare una variable Func y asigne una expresión lambda al variable.
3   // El método toma una cadena y la convierte a mayúsculas.
4   Func<string, string> selector = str => str.ToUpper();
5
6   // Crea una matriz de cadenas.
7   string[] words = { "naranja", "manzana", "articulo", "elefante" };
8   // Consulte la matriz y seleccione cadenas de acuerdo con el método de selector.
9   IEnumerable<String> aWords = words.Select(selector);
10
11  // Envíe los resultados a la consola.
12  foreach (String word in aWords)
13      Console.WriteLine(word);
14
```

Delegados(Action)

Los objetos de acción son como métodos de vacío, por lo que solo tienen un tipo de entrada. No se coloca ningún resultado en la pila de evaluación.

```
0 referencias
public static void Main()
{
    Action<string> messageTarget;

    if (Environment.GetCommandLineArgs().Length > 1)
        messageTarget = ShowWindowsMessage;
    else
        messageTarget = Console.WriteLine;

    Console.WriteLine("Hola, Mundo!");
}

1 referencia
private static void ShowWindowsMessage(string message)
{
    Console.WriteLine(message);
}
}
```

Inmutabilidad

Cualquier tipo que restrinja de alguna manera los cambios en su estado o en el estado de sus instancias puede describirse como inmutable en cierto sentido.

El tipo System.String es un tipo inmutable en el sentido de que el tamaño de la cadena, los caracteres y su orden no pueden cambiar.

El tipo System.MulticastDelegate, que es el padre de todos los tipos de delegados, es inmutable al igual que System.String.

Colecciones Inmutabilidad

Una colección inmutable es una colección de instancias que conserva su estructura todo el tiempo y no permite las asignaciones de nivel de elemento, mientras que sigue ofreciendo API para realizar mutaciones.

Las colecciones inmutables están diseñadas con dos objetivos en mente. En primer lugar, reutilizar la mayor cantidad de memoria posible, evitar la copia y reducir la presión sobre el recolector de basura El segundo objetivo es respaldar las mismas operaciones que ofrecen las colecciones mutables con complejidades de tiempo competitivas.

Pilas Inmutabilidad

Se representa como un puntero al elemento superior. De esta manera, puede empujar y hacer estallar elementos de una pila determinada sin cambiarla, mientras que al mismo tiempo comparte todos sus elementos con la pila resultante. Este diseño simple hace que la pila inmutable sea la colección inmutable más simple.

```
public static ImmutableStack<T> Create<T>();
public static ImmutableStack<T> Create<T>(T item);
public static ImmutableStack<T> Create<T>(params T[] items);
public static ImmutableStack<T> CreateRange<T>(IEnumerable<T> items);
```

Listas Inmutables

La estructura de datos de la lista es más compleja que la pila debido principalmente a la operación de indexación. La estructura de la lista ofrece recuperación, adición y eliminación de elementos en un índice específico. Usar una matriz, como se hace en la lista mutable <T>, sería razonable, pero, como se explicó anteriormente, sería ineficiente para una lista inmutable de propósito general.

```
ImmutableList<Int32> l1 = ImmutableList.Create<Int32>();
ImmutableList<Int32> l2 = l1.Add(1);
ImmutableList<Int32> l3 = l2.Add(2);
ImmutableList<Int32> l4 = l3.Add(3);
ImmutableList<Int32> l5 = l4.Replace(2, 4);
```

Colecciones

Las colecciones proporcionan una forma más flexible de trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con los que trabaja puede crecer y reducirse dinámicamente a medida que cambian las necesidades de la aplicación. Para algunas colecciones, puede asignar una clave a cualquier objeto que coloque en la colección para poder recuperar rápidamente el objeto usando la clave. Una colección es una clase, por lo que debe declarar una instancia de la clase antes de poder agregar elementos a esa colección.

Tipos de colecciones

NET proporciona muchas colecciones comunes. Cada tipo de colección está diseñado para un propósito específico.

Algunas de las clases de colección comunes se describen en esta sección:

- System.Collections.generic clases
- System.Collections.Concurrent clases
- Clases System.Collections

System.Collections.generic Classes

Puede crear una colección genérica utilizando una de las clases en el espacio de nombres System.Collections.generic .Una colección genérica es útil cuando todos los elementos de la colección tienen el mismo tipo de datos. Una colección genérica impone un tipo fuerte al permitir que se agregue solo el tipo de datos deseado.

System.Collections.Concurrent Classes

las colecciones en el espacio de nombres System.Collections.Concurrent proporcionan operaciones eficientes seguras para acceder a los elementos de la colección desde varios hilos.

Las clases en el espacio de nombres System.Collections.Concurrent deben usarse en lugar de los tipos correspondientes en los espacios de nombres System.Collections.Generic y System.Collections siempre que varios subprocesos accedan a la colección al mismo tiempo.

Clases System.Collections

Las clases en el espacio de nombres System.Collections no almacenan elementos como objetos de tipo específico, sino como objetos de tipo object

Siempre que sea posible, debe utilizar las colecciones genéricas en el espacio de nombres System.Collections.Generic o el espacio de nombres System.Collections. Concurrent en lugar de los tipos heredados en el System.Collections espacio de nombres.

Clases de uso frecuente

CLASES	DESCRIPCION
Dictionary<Tkey, Tvalue>	Representa una colección de pares clave / valor que se organizan en función de la clave.
List<T>	Representa una lista de objetos a los que se puede acceder por índice. Proporciona métodos para buscar, ordenar y modificar listas.
Queue<T>	Representa una colección de objetos primero en entrar, primero en salir (FIFO).
SortedList<Tkey, Tvalue>	Representa una colección de pares clave
Stack<T>	Representa una colección de objetos de último en entrar, primero en salir (LIFO).

Clases de uso frecuente

CLASES	DESCRIPCION
ArrayList	Representa una matriz de objetos cuyo tamaño aumenta dinámicamente según sea necesario.
HashTable	Representa una colección de pares clave / valor que se organizan en función del código hash de la clave.
Queue	Representa una colección de objetos primero en entrar, primero en salir (FIFO).
Stack	Representa una colección de objetos de último en entrar, primero en salir (LIFO).

Funciones de Orden Superior

Una función de orden superior es aquella que toma otra función como argumento o devuelve una función (o ambas).Esto se hace comúnmente con lambdas,

```
var results = data.Where(p => p.Items == 0);
```

La cláusula Where () podría recibir muchos predicados diferentes, lo que le da una flexibilidad considerable .Permite determinar que elementos queremos incluir y cuales excluir en la salida con una función Func<TSource, bool> que pasemos como predicado.

```
class Program
{
    public static Func<int, bool> esPar = x => x % 2 == 0;
    1 reference
    public static int TotalPares(List<int> listaNumeros, Func<int, bool> validaPar)
    {
        var total = 0;
        foreach (var item in listaNumeros)
        {
            if (validaPar(item))
                total++;
        }
        return total;
    }
    0 references
    static void Main(string[] args)
    {
        var numeros = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        Console.WriteLine($"El total de números pares entre 1 y 9 es {TotalPares(numeros,esPar)}");
    }
}
```

Tuples

Es una estructura de datos que nos permite almacenar hasta 8 valores diferentes, de diferentes tipos, que están relacionados de algún modo y usando una sola variable. Así, en una declaración rápida podemos crear una Tupla y utilizarla para cualquier necesidad

```
var tupla = new Tuple<int, String, DateTime>(1, "campusMVP", DateTime.Now);
```


Podemos usarlas accediendo a cada elemento a través de sus propiedades Item1, Item2, y sucesivos (hasta Item7). El octavo elemento se llama "rest" porque es el "resto" de la información que queramos añadir, y puede ser también una Tupla.
Se pueden crear también directamente infiriendo los tipos, lo cual es mucho más cómodo, usando el método estático Create de la clase Tuple,

```
var tupla2 = Tuple.Create(1, "campusMVP", DateTime.Now);
```

Cierres

- Cierra manualmente un objeto
- menú de salida que se controla para llamar explícitamente a Close

```
void fileExitMenuItem_Click(object sender, RoutedEventArgs e)
{
    // Close this window
    this.Close();
}
```


Recurrencias

OPERACIÓN	DESCRIPCION
GET	Obtiene la información de periodicidad para el identificador de periodicidad especificado.
LIST	Enumera todas las recurrencias.

Recurrencias – GET

GET: Obtiene la información de periodicidad para el identificador de periodicidad especificado.

```
namespace PollingService
{
    public sealed class ErrorResponse
    {
        public List<ErrorInfo> rows { get; set; }
        public string token { get; set; }
    }

    public sealed class  ErrorInfo
    {

        public string date { get; set; }
        public string errortype { get; set; }
        public string errorcode { get; set; }
        public string tankserial { get; set; }
        public int errorid { get; set; }

    }
}
```

Recurrencias LIST

LIST: Enumera todas las recurrencias.

```
private void button1_Click(object sender, EventArgs e)
{

    List<string> students = new List<string>();

    students.Add("Jenny");
    students.Add("Peter");
    students.Add("Mary Jane");

}
```

Funciones asíncronas

Como su nombre indica nos permite crear código que se va a ejecutar de una forma paralela. Proporciona una abstracción sobre el código asincrónico. Escribe el código como una secuencia de declaraciones, como siempre. Puede leer ese código como si cada declaración se completara antes de que comience la siguiente. El compilador realiza una serie de transformaciones porque algunas de esas declaraciones pueden comenzar a funcionar y devolver una tarea que representa el trabajo en curso.

Debemos priorizar la utilización de Task.WhenAll sobre la forma de esperar varias veces y esto es por varios motivos

- El código luce más limpio.
- Propaga los errores correctamente,

Funciones Parciales

brindan la capacidad de dividir la declaración de clases. Un problema común que se puede resolver con clases parciales es permitir a los usuarios modificar el código generado automáticamente sin temor a que sus cambios se sobrescriban si el código se vuelve a generar. Además, varios desarrolladores pueden trabajar en la misma clase o métodos.

```
2 referencias
public partial class PartialClass
{
    1 referencia
    public void ExampleMethod()
    {
        Console.WriteLine("Llamada al método desde la primera declaración.");
    }
}

2 referencias
public partial class PartialClass
{
    1 referencia
    public void AnotherExampleMethod()
    {
        Console.WriteLine("Llamada al método de la segunda declaración.");
    }
}
0 referencias
class Program
{
    0 referencias
    static void Main(string[] args)
    {
        PartialClass partial = new PartialClass();
        partial.ExampleMethod(); // outputs "Llamada al método de la primera declaración."
    partial.AnotherExampleMethod(); // outputs "Llamada al método de la segunda declaración."
}
```

PREPÁRATE
PARA SER EL
MEJOR

+ ENTREMIENTO
EXPERIENCIA

BIENVENIDOS.

GRACIAS
POR TU PARTICIPACIÓN

Por favor, bríndanos tus comentarios
y sugerencias para mejorar nuestros servicios.

GALAXY
TRAINING


GALAXY
TRAINING