



# PROGRAMACIÓN GENÉRICA

Estructura De Datos I

# CONTENIDO

1. Interfaces
2. Plantillas
3. Delegados
4. Predicados

# 1

## INTERFACES

Uso en programación genérica

# INTERFACES

- Es la definición de un grupo de funcionalidades relacionadas que una clase no abstracta o una estructura deben implementar.
- Pueden definir métodos estáticos, pero con su respectiva implementación.
- También permite definir implementaciones por defecto.
- No declara atributos, únicamente métodos.

# INTERFACES VS CLASES ABSTRACTAS

- Interfaces
  - No definen atributos
  - Una clase o estructura puede implementar más de una interfaz.
  - No implementan constructor.
- Clases abstractas
  - Pueden definir atributos
  - Una clase o estructura sólo puede heredar de una clase abstracta.
  - Pueden tener un constructor.



# 2

## PLANTILLAS

Programación genérica y  
conceptos

# PROGRAMACIÓN GENÉRICA



Programamos una vez, para múltiples soluciones

# PROGRAMACIÓN GENÉRICA

- Se utiliza cuando manejamos las mismas estructuras para resolver problemas similares.
- ¿Qué cambia en cada caso?
  - ▷ El tipo de dato



# PROGRAMACIÓN GENÉRICA

- Es un paradigma de programación centrado en los algoritmos y no en los datos
- Su postura fundamental es la generalización.

# PLANTILLAS

- Uno de los aportes de C++ a C
- Permite el envío de Tipos de Dato como parámetros a un programa
- Permite la creación de funciones, clases o estructuras genéricas

# PLANTILLAS

## Ventajas

- Generalización
  - ▷ Se puede reutilizar el código para varios tipos de dato
- Simplicidad
  - ▷ Permiten un fácil mantenimiento del código

## Desventajas

- Precauciones
  - ▷ Es necesario tomar medidas para evitar que el código falle con algunos tipos de datos
- Tiempo de compilación
- Necesidad de mayor abstracción

# PLANTILLAS

- Generan automáticamente el código de manejo para todas las clases utilizadas dentro del programa
- Se realiza en el tiempo de compilación



*El nombre del parámetro  
normalmente es una  $T''$*

# GENERIC

- Es la implementación de plantillas en .NET
- Es muy similar, pero tiene algunas diferencias

# GENERICOS - DIFERENCIAS

- El manejo genérico es realizado en tiempo de ejecución y no en tiempo de compilación
- No permite implementaciones personalizadas o parciales para algunos tipos de datos
- No permite especificar un tipo de dato default
- No se pueden utilizar parámetros de tipo plantilla

# GENERICIS

```
internal class LinearNode<T> where T : IComparable
{
    internal T Value { get; set; }
    internal LinearNode<T> Next { get; set; }

    internal LinearNode(T value)
    {
        Value = value;
        Next = null;
    }

    internal LinearNode()
    {
    }
}
```

Tipo de dato genérico

Precauciones en el manejo de tipos



# GENERIC

```
public class CustomLinkedList<T> : LinearDataStructureBase<T>, IEnumerable<T> where T: IComparable
{
    public bool IsEmpty { get; set; }
    public int Count { get; set; }
    private LinearNode<T> First { get; set; }
    private LinearNode<T> Last { get; set; }

    public CustomLinkedList()...

    public void AddMany(params T[] values)...

    public void Add(T value)...

    public void RemoveAt(int position)...

    public T GetPosition(int position)...

    protected override void Insert(T value)...

    protected override void Delete(int position)...

    protected override T Get(int position)...
```



# 3

## DELEGADOS

### Concepto

# DELEGADO

- Tipo de dato que representa una referencia a una firma de un método
- Un delegado se instancia con cualquier método de la misma firma, aunque el nombre sea distinto.
- El método se ejecuta a través de la instancia del delegado

# DELEGATE


```
internal void Sort(Delegate comparer, bool ascending)
{
    if (Comparer.IsComparer)
    {
        var Array = Array;
        var leftIndex = Array;
        var rightIndex = Array;

        var (left, right) = (Array, Array - 1);

        while (left < right)
        {
            leftIndex = left;
            rightIndex = right;

            if ((ascending && (int)comparer.DynamicInvoke(aElement.Value, bElement.Value) > 0)
                || (!ascending && (int)comparer.DynamicInvoke(aElement.Value, bElement.Value) < 0))
            {
                Swap(ref leftIndex, ref rightIndex);
            }

            leftIndex = leftIndex + 1;
            rightIndex = rightIndex - 1;
        }
    }
}
```



The diagram illustrates the arguments passed to the `DynamicInvoke` method. Two arrows point from the `aElement.Value` and `bElement.Value` expressions to the `DynamicInvoke` method call in the `if` statement. The `aElement.Value` and `bElement.Value` expressions are circled, and the `DynamicInvoke` method call is also circled.

# DELEGATE

```
public static Comparison<Student> SortByName = delegate (Student s1, Student s2)
{
    return s1.Name.CompareTo(s2.Name);
};

public static Comparison<Student> SortByLastName = delegate (Student s1, Student s2)
{
    return s1.LastName.CompareTo(s2.LastName);
};

public static Comparison<Student> SortByFaculty = delegate (Student s1, Student s2)
{
    return s1.Faculty.CompareTo(s2.Faculty);
};
```

Firma del delegado

```
studentList.SortDescending(Student.SortByName);
```

Uso en implementación

# 4

## PREDICADOS

### Concepto

# PREDICADO

- ▣ Delegado de un método que funciona como un criterio de filtrado

# FUNC<T, TResult>

- Es un tipo de delegado, principalmente utilizado para predicados, en el que la firma recibe como parámetro un objeto de tipo T y responde con un objeto de tipo TResult
- Los más comunes son
  - ▷ Func<T, bool>
    - ▷ Retorna true o false
  - ▷ Func<T, TResult>
    - ▷ Retorna un objeto de cualquier tipo



# PREDICATE<T>

- Tipo de dato proveniente de delegate
- La función debe tener de entrada un objeto de tipo T
- Debe retornar true o false
- Utilizada internamente para comparar
- Es muy similar a Func<T, bool>

# PREDICATE<T>

```
Predicate<Person> oscarFinder = (Person p) => { return p.Name == "Oscar"; };  
Predicate<Person> ruthFinder = (Person p) => { return p.Name == "Ruth"; };  
Predicate<Person> seventeenYearOldFinder = (Person p) => { return p.Age == 17; };  
  
Person oscar = people.Find(oscarFinder);  
Person ruth = people.Find(ruthFinder);  
Person seventeenYearOld = people.Find(seventeenYearOldFinder);
```

```
Predicate<int> pre = delegate(int a){ return a % 2 == 0; };
```

# EXPRESIONES LAMBDA

- Funciones “anónimas”
  - ▷ No se declaran, sólo se ejecutan
- Normalmente, trabajan como `Func<T, bool>` o `Func<T, TResult>`
- Consta de 3 partes
  - ▷ Declaración de parámetros
  - ▷ `=>` (se lee como “produce”)
  - ▷ Función

# EXPRESIONES LAMBDA

■ Ejemplo:  $x \Rightarrow x * x$

```
delegate int del(int i);  
static void Main(string[] args)  
{  
    del myDelegate = x => x * x;  
    int j = myDelegate(5); //j = 25  
}
```

# EXPRESIONES LAMBDA

- Uno de sus usos principales, son las funciones de LINQ
  - Language Integrated Query
- Nos permite hacer funciones similares a las de bases de datos, dentro de nuestro código

# EXPRESIONES LAMBDA

```
var ids = new List<int> { 5, 3, 7, 1, 2, 8, 9, 4 };
```

```
var filteredIds = ids.Where(x => x > 5);
```

▲ 1 of 2 ▼ (extension) `IEnumerable<int> IEnumerable<int>.Where<int> (Func<int, bool> predicate)`  
Filters a sequence of values based on a predicate.  
***predicate:** A function to test each element for a condition.*

```
var newStudentList = ids.Select(x => new Student { Id = x });
```

▲ 1 of 2 ▼ (extension) `IEnumerable<TResult> IEnumerable<int>.Select<int, TResult> (Func<int, TResult> selector)`  
Projects each element of a sequence into a new form.  
***selector:** A transform function to apply to each element.*