

# Apuntes de teoría .NET



# Teoría 1

## Common Language Runtima (CLR)

El CLR es el motor de ejecución (runtime) de .NET. Es un entorno virtual independiente de la arquitectura de hardware en el que se ejecutan las aplicaciones escritas con cualquier lenguaje .NET.

## Microsoft Intermediate Language (MSIL)

Cuando se compila una aplicación escrita en un lenguaje de .NET, el compilador genera código MSIL (o CIL). MSIL es un lenguaje similar a un código ensamblador pero de más alto nivel, creado para un hipotético procesador virtual que no está atado a una arquitectura determinada. De manera transparente el código MSIL se traduce al lenguaje nativo del procesador físico en tiempo de ejecución, por intermediación de un compilador bajo demanda "Just In Time" (JIT).



## Base Classes Library (BCL)

La BCL es la biblioteca de clases base de .NET que da soporte a infinidad de funcionalidades a través de las clases y otros tipos definidos en ella. Las clases en la BCL se organizan en espacios de nombres (namespaces) que agrupan a clases y a otros namespaces. Ejemplo: Los tipos integrados están en el espacio de nombres System

## Introducción a .NET

Para crear nuestra aplicación vamos a utilizar .NET CLI. La interfaz de la línea de comandos de .NET es un conjunto de herramientas multiplataforma que sirve para desarrollar, compilar, ejecutar y publicar aplicaciones .NET.

Todos los comandos comienzan con dotnet que es el controlador genérico de la CLI de .NET

## Directiva using

El nombre extenso de una clase (o de cualquier otro tipo) lleva como prefijo el espacio de nombres en la que se ha definido. El nombre extenso de la clase Console es System.Console porque está definida en el espacio de nombres System. Si usamos la directiva using System al inicio del código podemos referirnos a cualquier tipo definido en el espacio de nombres System sin anteponer el prefijo System.

El alcance de la directiva using es el archivo fuente donde se especifica.

### - Directivas using implícitas

El compilador agrega automáticamente un conjunto de directivas using en función del tipo de proyecto. En el caso de las aplicaciones de consola se incluye:

- using System;
- using System.IO;
- using System.Collections.Generic;
- using System.Linq;

- using System.Net.Http;
- using System.Threading;
- using System.Threading.Tasks;

## Tipos integrados

C# proporciona un conjunto estándar de tipos numéricos integrados para representar números enteros y valores de punto flotante. Además de los tipos numéricos, C# proporciona tipos integrados para representar expresiones booleanas, caracteres de texto, cadenas de texto y objetos. Todos los tipos integrados de C# son tipos de .NET y pueden referenciarse por el nombre del tipo en la plataforma o por un alias propio de C#.

Los tipos integrados son estructuras, salvo string y object que son clases. Están definidos en el espacio de nombres System, por lo tanto, si se utiliza la directiva using System es posible referirse al tipo System.Boolean simplemente como Boolean. Sin embargo, para el caso de los tipos integrados, lo usual es utilizar el alias definido para el lenguaje correspondiente.

## Teoría 2

### Common Type System (CTS)

Define un conjunto común de tipos orientado a objetos. Todo lenguaje de programación de .NET debe implementar los tipos definidos por el CTS. Los tipos de .Net pueden ser tipos de valor o tipos de referencia.



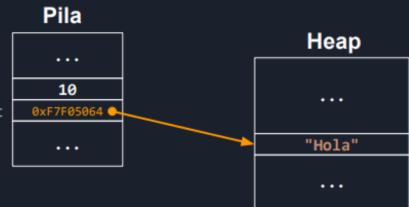
#### Sistema de tipos

**Tipos de valor:** El espacio reservado para la variable en la pila de ejecución guarda directamente el valor asignado.

**Tipos de referencia:** El espacio reservado para la variable en la pila de ejecución guarda la dirección en la memoria heap donde está el valor asignado.

## CÓDIGO

```
...
int n = 10;           // int es un tipo de valor
string st = "Hola"; // string es un tipo de referencia
...
```

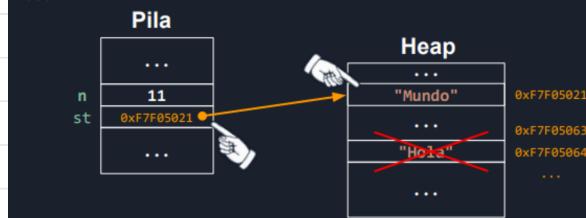


## CÓDIGO

```
...
int n = 10;
string st = "Hola";
n = 11;
st = "Mundo";
...

```

Asignar un nuevo valor a la variable `st` (tipo de referencia) implica conseguir espacio libre en la heap, colocar el nuevo valor en la nueva dirección escogida y modificar la memoria de la pila asociada a la variable `st` escribiéndole allí la nueva dirección



Nota: asignar un nuevo valor a una variable de tipo de valor, implica simplemente sobreescribir el valor anterior, mientras que asignar un nuevo valor a una variable de tipo de referencia implica generar una nueva dirección.

Common Type System adminte las cinco categorías de tipos siguientes:

- Estructuras
- Enumeraciones

Tipos de valor

- Clases
- Delegados
- Interfaces

Tipos de referencia

## Conversión de tipos

Conversiones implícitas

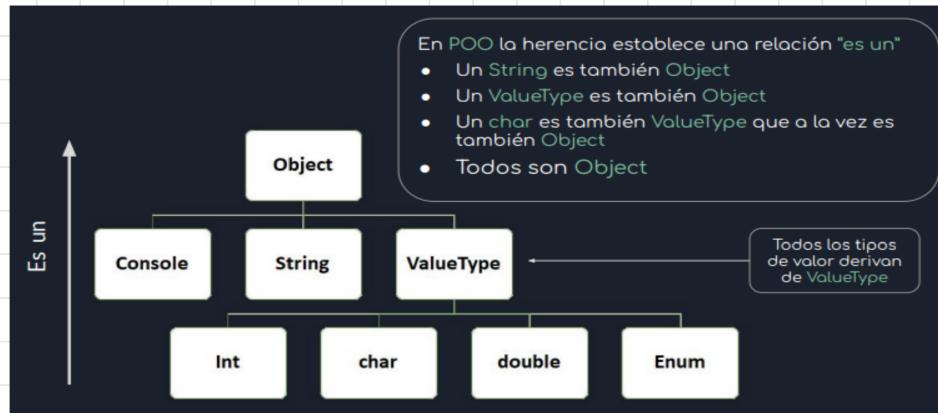
```
int i = b; ←
double y = i; ←
```

Conversiones Explícitas

```
short j = (short)i;
i = (int)x;
```

La conversión de tipo implícita se realiza cuando la operación es segura. En otro caso, se requiere el consentimiento del programador quien debe hacerse responsable de la seguridad de la operación.

**Sistema unificado de tipos:** Todos los tipos de datos derivan directa o indirectamente de un tipo base común: la clase System.Object. Esto también es aplicable a los tipos de valor (conversiones boxing y unboxing)



Consecuencia de tener un Sistema unificado de tipos: aunque C# es un lenguaje fuertemente tipado, debido a la jerarquía de tipos y a la relación "es un", las variables de tipo object admite valores de cualquier tipo. ¿Cómo es posible asignar a una variable de tipo referencia una de tipo valor? **Las conversiones boxing y unboxing lo hacen posible.**

**Boxing y Unboxing:** Las conversiones boxing y unboxing permiten asignar variables de tipo de valor a variables de tipo de referencia y viceversa.

- **Boxing:** Cuando una variable de algún tipo de valor se asigna a una de tipo de referencia, se dice que se le ha aplicado la conversión boxing

- **Unboxing:** cuando una variable de algún tipo de referencia se asigna a una de tipo de valor, se dice que se le ha aplicado la conversión unboxing

### El valor null

Si queremos que una variable de un tipo T admita, además de todos los valores propios de T, el valor especial null (es decir que sea nullable) usamos "?" en la declaración de la variable.

Ejemplo: int? numero = null; es una instrucción correcta porque la variable "numero" acepta el valor null

## Valor null en tipos de valor

Dado que cualquier secuencia de bits asociada en la pila a una variable de tipo de valor representará un dato válido, para implementar un tipo de valor que admita null se utilizó una estructura con dos propiedades (HasValue y Value)

HasValue: Retorna un boolean, true si la variable tiene un valor distinto a null, false si el valor es null

Value: Retorna el valor de la variable. Si el valor es null retorna un valor por defecto según el tipo de dato

## Operador is

Con el operador is podemos consultar por el tipo de una expresión o contenido de una variable. Devuelve un bool.

### Semántica del operador is :

```
object o = 1;
Console.WriteLine(o is int); → True
Console.WriteLine(o is ValueType); → True
Console.WriteLine(o is object); → True
Console.WriteLine(o is string); → False
Console.WriteLine(3 * 1.1 is double); → True
```

## Operador as

Se puede utilizar con los tipos que admiten null.

El operador as, al igual que una expresión cast, se utiliza para realizar una conversión explícita hacia un tipo que admite null. Cuando no se puede llevar a cabo la conversión el operador as devuelve null mientras que una expresión cast lanza una excepción.

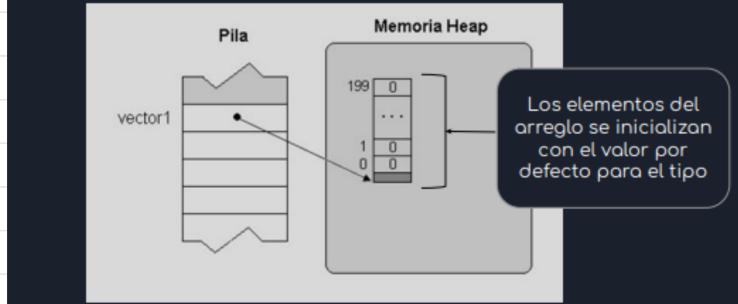
## Arreglos

Los arreglos son de tipo referencia. Pueden tener varias dimensiones, el numero de dimensiones se denomina Rank.

El número total de elementos de un arreglo se llama longitud del arreglo (length)

- Como se ve un arreglo de una dimensión (vector) en la pila y la memoria heap

```
int[] vector1;
vector1 = new int[200];
```



## Clase String

- Es un tipo de referencia, sin embargo la comparación no es por dirección de memoria. Se ha redefinido el operador == para realizar una comparación lexicográfica. Tiene en cuenta mayúsculas y minúsculas.
- Los strings son de inmutables (no se pueden modificar caracteres individuales)
- Acceso a los elementos: [ ]
- Primer elemento: Índice cero

## Clase StringBuilder

Símil a un string de lectura/escritura. Definida en el espacio de nombre System.Text

Métodos adicionales:

- Append -Insert -Remove -Replace ---etc

Permite modificar caracteres individuales. Se tiene que instanciar un objeto de este tipo y luego, accediendo a sus caracteres (por indices) a través de corchetes, podemos modificar el carácter que queramos, sin necesidad de crear un nuevo espacio en memoria

# Tipos enumerativos

Un enumerativo es un conjunto de constantes con nombre que representan valores subyacentes, típicamente de tipo entero. Las enumeraciones se utilizan para definir un conjunto de valores discretos y con nombre que una variable puede tener, mejorando así la legibilidad y la mantenibilidad del código.

```
class Program
{
    static void Main()
    {
        DiaDeSemana diaDeReunion = DiaDeSemana.Martes;
        for (DiaDeSemana d = DiaDeSemana.Lunes; d <= DiaDeSemana.Viernes; d++)
        {
            if (d == diaDeReunion)
            {
                Console.WriteLine("El " + d + " es día de reunión");
            }
        }
    }

    enum DiaDeSemana
    {
        Domingo, Lunes, Martes,
        Miércoles, Jueves,
        Viernes, Sábado
    }
}
```

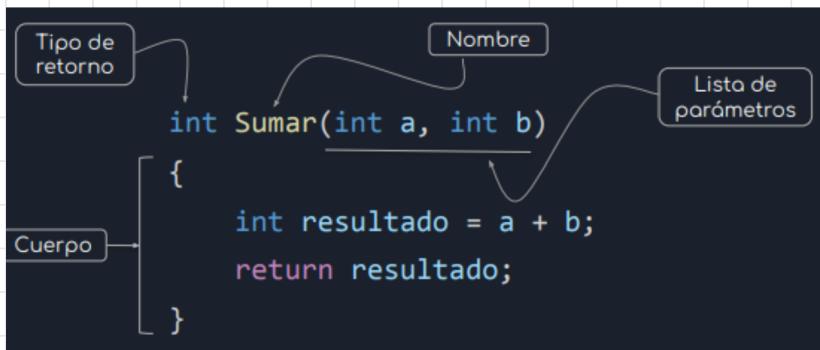
Instrucciones de nivel superior

Declaración de tipo

Las instrucciones de nivel superior deben preceder a las declaraciones de tipos  
Sin embargo es aconsejable dejar en el archivo Program.cs sólo instrucciones de nivel superior y declarar cada tipo en su propio archivo fuente

## Métodos

Bloque con nombre de código ejecutable. Puede invocarse desde diferentes partes del programa, e incluso desde otros programas



Si el método no devuelve ningún valor, se especifica void como tipo de retorno. En ese caso return es opcional.

## Pasaje de parámetros



Por valor

Por referencia

De salida

- **Por valor:** Recibe una copia del valor pasado como parámetro

- **Por referencia:** Recibe una dirección. En la invocación también se debe utilizar "ref". La variable pasada por ref debe estar inicializada.

También existe ref readonly, el valor de ese tipo no puede modificarse, es de sólo lectura

- **De salida:** También recibe una dirección. La variable pasada por out puede estar sin inicializar. En la invocación también se debe utilizar out. El método invocado es el responsable de establecer el valor del parámetro de salida.

### Uso de la palabra clave params

Permite que un método tome un número variable de argumentos. El tipo declarado del parámetro params debe ser un arreglo unidimensional

```
int[] vector = { 1, 2, 3 };
Imprimir(vector);
Imprimir();
Imprimir(2, 4, 8, 5, 5);
```

Se puede pasar un vector de enteros o una lista de cero o más enteros

```
static void Imprimir(params int[] vector)
{
    foreach (int i in vector)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine("Ok");
}
```



## Métodos con forma de expresión (expression-bodied methods)

Para los casos en que el cuerpo de un método pueda escribirse como una sola expresión, es posible utilizar una sintaxis simplificada.

Esta sintaxis no está limitada a métodos que devuelven void, se puede utilizar con cualquier tipo de retorno

Ejemplo:

```
void Imprimir(string st)
{
    Console.WriteLine(st);
}
```

Puede escribirse como:

```
void Imprimir(string st) => Console.WriteLine(st);
```

Ejemplo

```
int Suma(int a, int b)
{
    return a + b;
}
```

Observar que no va la sentencia return

Puede escribirse como:

```
int Suma(int a, int b) => a + b;
```

## Teoría 3

### Arreglos de dos dimensiones

Forma de cargar una matriz con mod/div



```
int[,] mat = new int[3, 4];
for (int i = 0; i <= 11; i++)
{
    mat[i / 4, i % 4] = i;
}
```

- Matriz

```
int[,] matriz = new int[,]
{
    {1,2,3,4},
    {5,6,7,8},
    {9,10,11,12}
};
```

1	2	3	4
5	6	7	8
9	10	11	12

- Acceso

```
matriz[2, 2] = matriz[1, 1] + matriz[1, 2];
```

1	2	3	4
5	6	7	8
9	10	13	12

### Inferencia de tipos - palabra clave var

La palabra clave var en la declaración de una variable local con inicialización (<> null) indica que el tipo de la misma es inferido por el compilador en función de la inicialización.

Una vez inferido el tipo de una variable por el compilador queda fijo e inmutable (no es un tipo dinámico)

```

var i = 15;
i = 140;
i = 13.2;

```

El tipo inferido de `i` es `int`

Error de compilación, no se puede convertir implícitamente el tipo `double` en `int`

Básicamente, una vez que toma un tipo, no se puede cambiar. No podemos asignarle un valor de otro tipo que no sea el inferido.

**Tipos anónimos:** La inferencia de tipos permite instanciar objetos de tipos anónimos. Una forma conveniente de encapsular un conjunto de propiedades de solo lectura en un solo objeto sin tener que definir explícitamente un tipo primero.

X e Y son variables de tipos anónimos distintos

```

var x = new { Nombre = "Juan", Edad = 28 };
var y = new { Alto = 12.4, Ancho = 11, Largo = 20 };
Console.WriteLine("Nombre de x: " + x.Nombre);
Console.WriteLine("Ancho de y: " + y.Ancho);

```

**Tipo dynamic:** Una variable declarada de tipo dynamic admite la asignación de elementos de distintos tipos durante la ejecución

El tipo dynamic funciona como si fuese el tipo object pero el compilador omite la verificación de tipos, simplemente supone que la operación es válida. Esto no nos previene de errores en tiempo de ejecución.

Tampoco son necesarias las conversiones explícitas.

```

dynamic dy = "hola mundo!";
Console.WriteLine(dy.GetType());
Console.WriteLine(dy);
dy = 140;
Console.WriteLine(dy.GetType());
Console.WriteLine(dy);

```

```

System.String
hola mundo!
System.Int32
140

```

## Strings de formato compuesto y strings interpolados

```

st = string.Format("Es un {0} año {1}", marca, modelo);

```

Made with Goodnotes  
Esta es una cadena de formato compuesto. Es un string con marcadores de posición indexados

vector de objetos

## Las cadenas

interpoladas también utilizan elementos de formato. Son mas legible y cómodas de utilizar que los strings de formato compuesto. Las cadenas interpoladas llevan antepuesto el símbolo \$. Los elementos de formato en las cadenas interpoladas también permite expresiones

```
int ancho = 30;  
Console.WriteLine($"ancho = {ancho} y alto = {20}");  
Console.WriteLine($"superficie = {ancho * 20 / 2}");  
Console.WriteLine($"Saludo: {"Hola"+ " Mundo"}");
```

```
ancho = 30 y alto = 20  
superficie = 300  
Saludo: Hola Mundo
```

El método ToString() definido en los tipos numéricos también acepta un parámetro que es una máscara de formato.

```
double r = 2.417;  
string st = r.ToString("0.00");  
st queda asignado con el string "2,42"
```

## Dificultades de los arreglos:

- > Incrementar la longitud del arreglo
- > Reducir la longitud del arreglo
- > Insertar un elemento en cualquier posición
- > Borrar un elemento reduciendo la longitud del arreglo
- > Acceder a los elementos por medio de un índice no entero

## Colecciones

Las colecciones, al igual que los arreglos, gestionan un conjunto de elementos pero lo hacen de una manera especial. Se pueden considerar arreglos especializados, existen distintos tipos de colecciones especializadas en distintas tareas

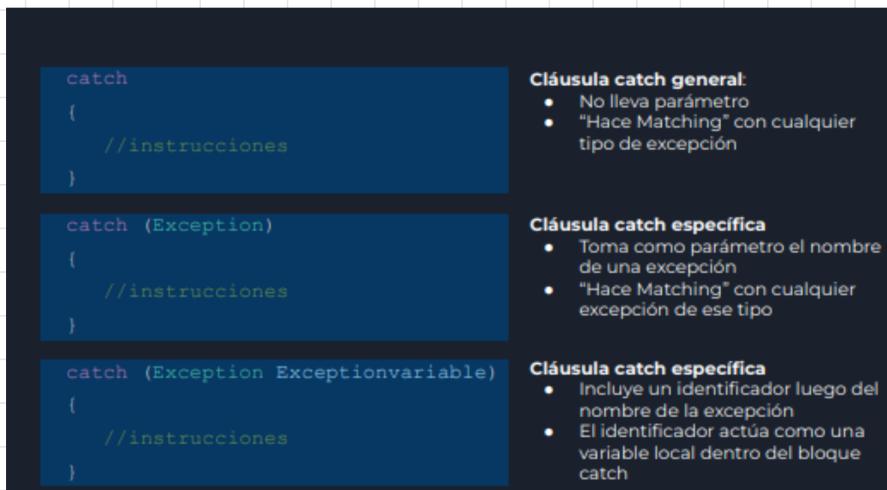
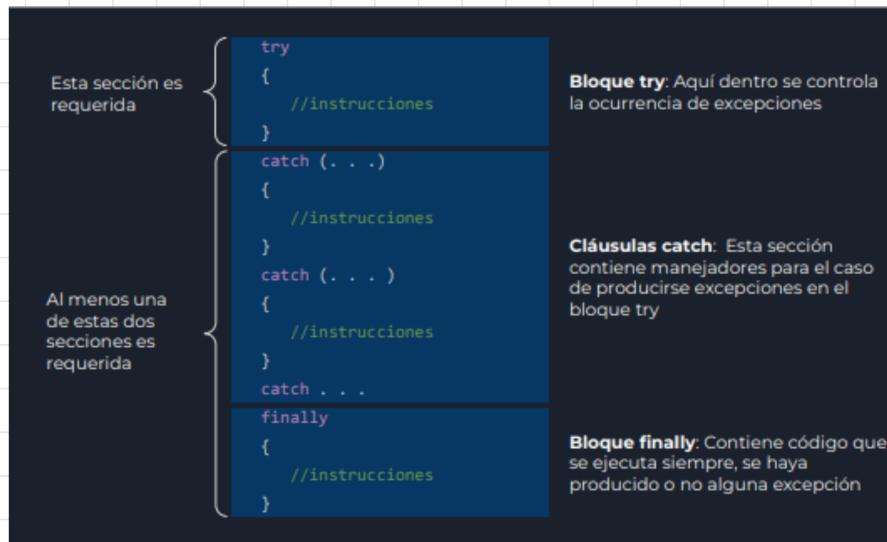
### Colecciones genéricas:

- List <T>. Similar a un vector de elementos de tipo T pero con ciertas facilidades como por ejemplo ajustar su tamaño dinámicamente
- Stack <T>. Pila de elementos de tipo T.
- Queue <T>. Cola de elementos de tipo T.

-Dictionary <TKey,TValue>. Colección de pares (clave, valor). Recuperar un valor usando su clave, es muy rápido porque se implementa como una tabla hash.

## Manejo de excepciones

Las excepciones son errores en tiempo de ejecución. Ejemplos: Intentar dividir por cero, escribir un archivo de solo lectura, referencias a null, acceder a un arreglo con un índice fuera de rango válido, etc



El bloque finally se ejecuta siempre antes de finalizar el try/catch independientemente de la ejecución o no de alguna cláusula catch.

Se ejecuta aún si se alcanza una sentencia return en el bloque try o alguno de los bloques catch

## Propagación de excepciones

Si método1 invoca a método2 y dentro de este último se produce una excepción que no es manejada, ésta se propaga a método1. Desde la perspectiva de método1, la invocación de método2 es la instrucción que genera la excepción

### Lanzar una excepción

- Para ello se utiliza el operador **Throw**
- uso:
  - `throw e`
  - Dentro de un bloque **catch**, relanza la excepción corriente

Lanza la excepción e, siendo e una instancia de una clase derivada de `System.Exception`

Si queremos lanzar una excepción, verificamos cuando lanzarla y hacemos:

`throw new ArgumentNullException("st")`

Es solo un ejemplo, puede ser cualquier otra excepción

Es posible crear nuestros propios tipos de excepciones, pero también podemos lanzar una excepción genérica con un mensaje personalizado:

```
throw new Exception("msg personalizado");
```

## Teoría 4

### Programación orientada a objetos

Es una manera de construir Software. Es un paradigma de programación. Propone resolver problemas de la realidad a través de identificar objetos y relaciones de colaboración entre ellos. El objeto y el mensaje son sus elementos fundamentales

La POO en .Net está basada en las clases, una clase describe el comportamiento (métodos) y los atributos (campos) de los objetos que serán instanciados a partir de ella

## Clases

Todos los métodos que definimos dentro de una clase son miembros de esa clase

Los miembros de una clase pueden ser:

- De instancia: pertenecen al objeto
- Estáticos: pertenecen a la clase

Sintaxis:

```
class <NombreDeLaClase>
{
    <Miembros>
}
```

### Miembros de instancia:

- Campos      -Métodos      -Constructores      -Constantes
- Propiedades    -Indizadores    -Finalizadores (o Destructores)
- Eventos       - Operadores    - Tipos anidados

### Campos de instancia

Un campo o variable de instancia es un miembro de datos de una clase.

Cada objeto instanciado de esa clase tendrá su propio campo de instancia con un propio valor

**Sintaxis:** Se declara dentro de una clase con la misma sintaxis con que declaramos variables locales dentro de los métodos

```
<tipo> <variable>;
```

Sin embargo, los campos se declaran fuera de los métodos

Los miembros de una clase son privados de la clase por defecto. Los campos Marca y Modelo son privados, por lo tanto sólo se pueden acceder desde el código de la clase Auto, pero no es posible hacerlo desde fuera de esta clase

## Métodos de instancia

Los métodos de instancia permiten manipular los datos almacenados en los objetos. Implementan el comportamiento de los objetos, dentro de los métodos de instancia se pueden acceder a todos los campos del objeto, incluidos los privados.

## Constructores de instancia

Un constructor de instancia es un método especial que contiene código que se ejecuta en el momento de la instanciación de un objeto. Habitualmente se utilizan para establecer el estado del nuevo objeto por medio del pasaje de argumentos.

**Sintaxis:** Se define como un método sin valor de retorno con el mismo nombre que la clase

```
<modificadorDeAcceso> <NombreDelTipo>(<parámetros>)
{
    ...
}
```

No debe ser privado si se desea crear instancias fuera de la Clase

Ejemplo: Constructor de la clase `Auto`

```
public Auto(string marca, int modelo)
{
    ...
}
```

Mismo nombre que la clase  
No hay tipo de retorno

para que pueda ser invocado desde fuera de la clase

Si no se define un constructor explícitamente, el compilador agrega uno sin parámetros y con cuerpo vacío.

Si se define un constructor explícitamente, el compilador ya no incluye el constructor por defecto.

**Sobrecarga:** Es posible tener más de un constructor en cada clase (sobrecarga de constructores) siempre que defiera en alguno de los siguientes puntos:

- La cantidad de parámetros
- El tipo y el orden de los parámetros
- Los modificadores de los parámetros

Los métodos también se pueden sobrecargar, con estas mismas consideraciones

## Arquitectura modular

### Arquitectura NO modular

Único módulo ejecutable que implementamos en un único proyecto

### Arquitectura modular

Varios módulos, ejecutables y bibliotecas de clases, que implementaremos como una solución con varios proyectos

## Biblioteca de clases (.dll)

Las bibliotecas de clases permiten dividir funcionalidades útiles en módulos que pueden usar varias aplicaciones. .Net maneja el concepto de "solución" para agrupar varios proyectos. Por ejemplo, podríamos crear una solución con dos proyectos, uno de ellos será un ejecutable, el otro puede ser una biblioteca de clases. Tanto los ejecutables, como a las bibliotecas de clases reciben el nombre de ensamblados en .Net

## Teoría 5

### Miembros estáticos

Los miembros estáticos son miembros que pertenecen a la propia clase (o tipo) en lugar de a un objeto determinado (instancia) de la misma.

```
namespace Teoria5;
class Cuenta
{
    public static void ImprimirResumen()
        => Console.WriteLine("No hay datos");
}
```

Método estático

Como acceder

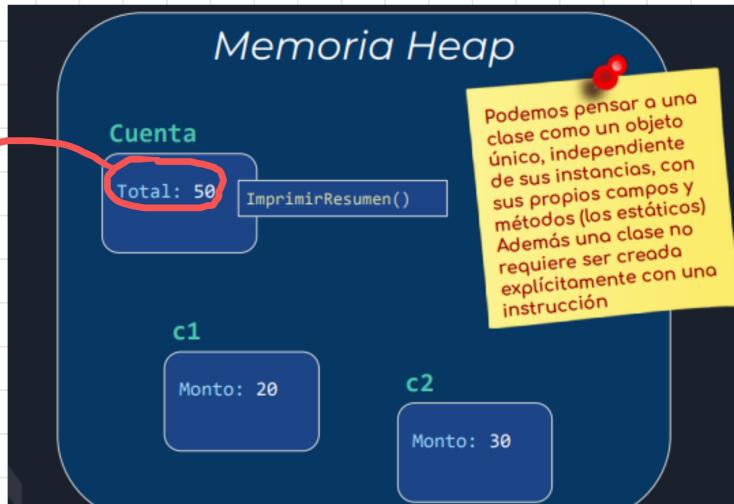
```
<NombreClase>.<Miembro>
```

por ejemplo:

```
Cuenta.ImprimirResumen()
```

## Campos estáticos

Un campo estático de una clase es una variable accesible a través de la clase que fue definido (NO a través de las instancias). Es una única variable compartida por todas las instancias de esa clase, incluso por objetos de otras clases en caso de no ser privada



En los métodos estáticos sólo están visibles los campos y métodos estáticos de la clase

## Constructores estáticos

Un constructor estático se declara como uno de instancia pero con el modificador static. No se puede invocar explícitamente, es invocado por el runtime de .Net una única vez cuando se carga la clase, por lo tanto:

- > No se pueden tener parámetros ni modificadores de acceso.
- > No pueden sobrecargarse (sólo puede definirse un constructor estático por clase)

El runtime de .Net no garantiza cuando se ejecutará un constructor estático, ni en qué orden se ejecutarán los constructores estáticos de diferentes clases. Sin embargo, lo que está garantizado es que el constructor estático se ejecutará antes de que nuestro código haga referencia a la clase

## Clases estáticas

Las clases estáticas llevan el modificador static en su declaración, solo pueden contener miembros estáticos. No es posible instanciar objetos de una clase estática. Ejemplos de clases estáticas: Console, File, Directory, Math, etc.

Las clases estáticas por lo general constituyen "clases de utilidades" (utility classes), agrupando cierta funcionalidad que se expone completamente como miembros de nivel de clase (estáticos). Un claro ejemplo es la clase Math

## Campos constantes (const)

Son valores inmutables que no cambian durante la vida del programa, se conocen en tiempo de compilación. Se declaran con el modificador const y deben inicializarse cuando se declaran. Son siempre implícitamente estáticas, sin embargo no se usa el modificador static

La expresión que se asigna a una constante es computada por el compilador, por lo tanto:

- Es una expresión simple
- No puede referir a ninguna variable (todas las variables, aún las estáticas se inicializan en tiempo de ejecución)
- No puede implicar la ejecución de código del programa.

## Campos readonly

Con los campos readonly se obtiene un efecto similar al que tienen los campos constantes pero sin sus restricciones. Se identifican con el modificador readonly y pueden ser de instancia o estáticos.

Sólo pueden asignarse en su declaración o dentro de un constructor. Se asignan en tiempo de ejecución como cualquier variable, por lo tanto no están restringidos a un conjunto de tipos u operaciones simples como en el caso de las constantes.

## Encapsulamiento

El encapsulamiento es uno de los pilares de la programación orientada a objetos, es la capacidad del lenguaje para ocultar detalles de implementación hacia fuera del objeto. En estrecha relación con la noción de encapsulamiento está la idea de la protección de datos. Idealmente, el estado de los objetos debería especificarse usando campos privados

Usar getters y setters no es lo indicado en la plataforma .Net

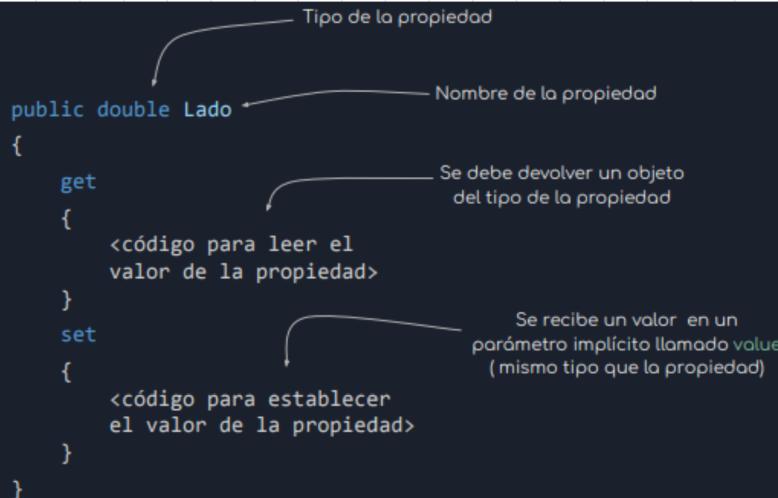


Se deben usar propiedades

Una propiedad integra conceptos de campo y método al mismo tiempo. Externamente se asigna y lee como si fuese un campo, internamente se codifican dos bloques de código:

- bloque get: se ejecuta cuando se lee la propiedad
- bloque set: se ejecuta cuando se escribe la propiedad

A los bloques get y set se los llama descriptores de acceso (accesors en inglés)



```
public double Lado
{
    get
    {
        <código para leer el valor de la propiedad>
    }
    set
    {
        <código para establecer el valor de la propiedad>
    }
}
```

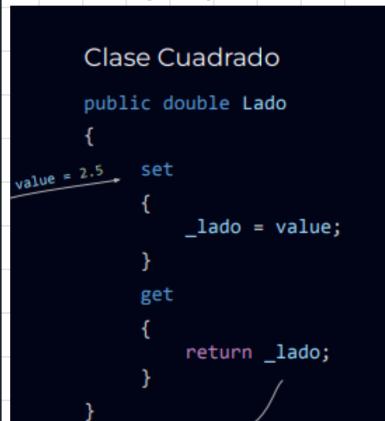
Tipo de la propiedad: public double Lado

Nombre de la propiedad: Lado

Se debe devolver un objeto del tipo de la propiedad: <código para leer el valor de la propiedad>

Se recibe un valor en un parámetro implícito llamado value (mismo tipo que la propiedad): <código para establecer el valor de la propiedad>

Ejemplo



```
Clase Cuadrado
public double Lado
{
    value = 2.5
    set
    {
        _lado = value;
    }
    get
    {
        return _lado;
    }
}
```

Una propiedad que implementa sólo el bloque get es una propiedad de sólo lectura. Una propiedad que implementa sólo el bloque set es una propiedad de sólo escritura (no es aconsejable usar esta)

Si el cuerpo de un descriptor de acceso consta de una sola expresión puede utilizarse la sintaxis alternativa (*expression bodied member*)

```
public double Lado  
{  
    get  
    {  
        return _lado;  
    }  
    set  
    {  
        _lado = value;  
    }  
}
```

Equivalente

```
public double Lado  
{  
    get => _lado;  
    set => _lado = value;  
}
```

Las propiedades públicas siempre son preferibles a los campos públicos porque, al ser miembros de funciones (no de datos como los campos), pueden procesar la entrada y la salida lo que permite establecer controles sobre los valores de la propiedad

Si la clase que modificamos es utilizada por otro programa, al cambiar un campo por una propiedad debemos volver a compilar también el otro programa.

Esto no sucede si desde el principio se codifica una propiedad. Al cambiar su implementación no es necesario recompilar otros programas que acceden a ella.

A menudo, una propiedad public se encuentra asociada a un campo privado (backing field)

Para facilitar la tarea del programador, se introdujeron las propiedades auto-implementadas.

Con ellas es posible declarar la propiedad sin declarar el campo asociado, el compilador crea un campo oculto, no está accesible para el programador, que asocia a la propiedad auto-implementada.

No hay excusas para seguir utilizando campos públicos. Sólo agregando `{get;set;}` los convertimos en propiedades auto-implementadas

```
class Persona  
{  
    public string Nombre;  
    public int Edad;  
    public string Email;  
}
```

Made with Goodnotes

```
class Persona  
{  
    public string Nombre {get;set;}  
    public int Edad {get; set;}  
    public string Email {get;set;}  
}
```

Clase con tres  
propiedades públicas  
auto-implementadas

Las propiedades  
también pueden ser  
estáticas

## Indizadores

Un indizador es una definición de cómo aplicar el operador ([ ]) a los objetos de una clase. A diferencia de los arreglos, los índices que se les pase entre corchetes no están limitados a los enteros, pudiéndose definir varios indizadores en una misma clase siempre y cuando cada uno tome un número o tipo de índices diferentes (sobrecarga). Los indizadores son sólo de instancia, no pueden definirse indizadores estáticos.

### Indizadores - Sintaxis

```
public Persona this[<lista de indices>]
{
    get
    {
        código que retorna el elemento
        según la <lista de índices>
    }
    set
    {
        código que establece el elemento
        según la <lista de índices>
    }
}
```

El nombre del indizador es siempre this

índices que identifican un elemento

Se recibe el elemento en un parámetro隐式 llamado value del mismo tipo que el indizador

Tipo del indizador

```
class Familia {
    ...
    public Persona? this[int i]
    {
        get
        {
            if (i == 0) return Padre;
            else if (i == 1) return Madre;
            else if (i == 2) return Hijo;
            else return null;
        }
    }
}

using Teoria5;
Familia f = new Familia();
f.Padre = new Persona("Juan", 50);
f.Madre = new Persona("María", 40);
f.Hijo = new Persona("José", 15);
for (int i = 0; i < 3; i++)
{
    f[i]?.Imprimir();
}
```

## Teoría 6

### Herencia

La herencia permite crear clases que reutilizan, extienden y modifican el comportamiento definido en otras clases. La clase cuyos miembros se heredan se denomina **clase base** y las clases que heredan esos miembros se denominan **clases derivadas**. Una clase derivada sólo puede tener una clase base directa, pero la herencia es transitiva.

Cuando tenemos clases que comparten atributos y/o comportamiento, es posible generalizar el diseño colocando las características comunes en una superclase.

Una clase derivada obtiene implícitamente todos los miembros de la clase base, salvo sus constructores y sus finalizadores.

Nota: Todas las clases (menos object) en la plataforma .Net derivan directa o indirectamente de la clase object

Luego, la clase que derivada al invalidar el método debe utilizar la palabra clave override. Así indicamos que el método está siendo invalidado, si no ponemos esta palabra clave estaríamos ocultando el método en lugar de invalidarlo.



Las clases derivadas pueden invalidar los métodos heredados para proporcionar una implementación alternativa. El método de la clase base debe marcarse con la palabra clave virtual

## Acceso a miembros de la clase base

La palabra clave base se utiliza para obtener acceso a los miembros de la clase base desde una clase derivada. Se utiliza en dos situaciones:

- Para invocar a un método (u otro miembro) de la clase base
- En el encabezado de un constructor para indicar el constructor de la clase base que se debe invocar

## Invalidación de propiedades

Las propiedades, al igual que los métodos, pueden ser redefinidas (invalidadas) en las clases derivadas

Para eso también tenemos que marcar la propiedad con la palabra clave virtual

- Las **clases** no pueden ser más accesibles que su **clase base**
- Ningún **miembro** puede ser más accesible que su **tipo** o el **tipo** de sus **parámetros**, **índices** o **valor de retorno**

## Clases abstractas

El propósito de una clase abstracta es proporcionar una definición común de una clase base que hereden múltiples clases derivadas.

No se pueden crear instancias de una clase abstracta. Se señalan con el modificador abstract

Una clase abstracta puede tener métodos, propiedades, indizadores y eventos abstractos.

También puede tener miembros no abstractos.

Los miembros abstractos no tienen implementación, se escriben sin el cuerpo, y llevan el modificador abstract. Los miembros abstractos son también virtuales y deben implementarse (override) en las subclases concretas.

## Finalizadores (destructores)

Los finalizadores se usan para "hacer limpieza" cuando el recolector de elementos no utilizados libera memoria.

El garbage collector comprueba periódicamente si hay objetos no referenciados por ninguna aplicación. En tal caso llama al finalizador (si existe) y libera la memoria que ocupaba el objeto.

En las aplicaciones de .Net framework, cuando se cierra el programa también se llama a los finalizadores

No se puede llamar a los finalizadores, se invocan automáticamente. Puede haber un solo finalizador por clase, no puede tener ni parámetros ni modificadores. Los finalizadores no se heredan

El programador no controla cuándo se llama al finalizador, ni el orden de las instancias finalizadas

## Polimorfismo

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después del encapsulamiento y la herencia

Made with Goodnotes

- Sintaxis:

```
class MiClase
{
    ~MiClase() // Finalizador o destructor
    {
        // código de limpieza...
    }
}
```

- Luego de ejecutarse el código de un finalizador se invoca implícitamente al finalizador de su clase base. Es decir que se ejecutan en cadena todos los finalizadores de manera recursiva desde la clase más derivada a la menos derivada.



```
object o;  
o = 1;  
o = "ABC";  
  
Automotor a;  
  
a = new Auto("Ford", 2000, TipoAuto.Deportivo);  
  
a = new Colectivo("Mercedes", 2010, 20);
```

**o y a** son objetos polimórficos.  
En tiempo de ejecución va a  
ocurrir polimorfismo (van a  
adoptar distintas formas de tipos)

Los miembros virtuales, la  
invalidación y el enlace dinámico  
permiten aprovechar el  
polimorfismo. (Las interfaces  
también).

## Interfaz polimórfica

La interfaz polimórfica de una clase base es el conjunto de sus miembros virtuales. Un diseño adecuado de la jerarquía de clases permitirá tomar ventaja del polimorfismo. Como ejemplo tenemos el famoso `ToString()`. Cada clase derivada lo implementa de forma distinta invalidando este método de su superclase. Entonces cuando instanciamos objetos, cada uno adopta su tipo y al invocar al método cada uno lo hace de la forma en la que esté definido.

## Principio open/close

El principio open/close (el segundo de los principios SOLID) establece:

Una entidad de software debe ser abierta para su extensión, pero cerrada para su modificación

El polimorfismo contribuye a que podamos cumplir con este principio

## Teoría 7

### Interfaces

Es un tipo de referencia que especifica un conjunto de funciones sin implementarlas. Pueden especificar métodos, propiedades, indizadores y eventos de instancia, sin implementación. En lugar del código que los implementa llevan un punto y coma. Por convención comienzan con la letra "I" mayúscula.

UNA INTERFAZ NO PUEDE CONTENER CAMPOS DE INSTANCIA,  
CONSTRUCTORES DE INSTANCIA NI FINALIZADORES

## Declaración de una interfaz

```
public interface IMiInterface  
{  
    public void UnMetodo();  
}
```

Los miembros de las interfaces son públicos de manera predeterminada.

Las clases derivan de otras clases y opcionalmente implementan una o más interfaces.

Si una clase implementa una interfaz debe implementar todos los miembros de la interfaz que no tienen implementación predeterminada. Es posible definir y utilizar variables de tipo interfaz:

```
Rombo r1 = new Rombo();  
Figura r2 = new Rombo();  
IAgrandable r3 = new Rombo();  
IIImprimible r4 = new Rombo();
```

### NOTA

No es posible crear una instancia de una interface

También es posible utilizar tipos interfaz para declarar propiedades, indizadores y métodos.

```
IIImprimible Elemento {get;set;}  
IIImprimible this [int index]...  
void EstablecerElemento(IIImprimible e) ...  
IIImprimible ObtenerElemento() ...
```

## File System

La BCL incluye todo un espacio de nombres llamado System.IO especialmente orientado al trabajo con archivos. Entre las clases más utilizadas de este espacio están:

--> Path --> DirectoryInfo --> FileInfo  
--> Directory --> File

### La clase Path:

Incluye un conjunto de miembros estáticos diseñados para realizar cómodamente las operaciones más frecuentes relacionadas con rutas y nombres de archivos

## Las clases DirectoryInfo, FileInfo, Directory y File

Para trabajar con archivos se utilizan objetos de la clase FileInfo y para trabajar con directorios objetos de la clase DirectoryInfo

Las clases File y Directory que sólo tienen métodos estáticos (al igual que Path) son útiles para realizar tareas sencillas. No requieren la creación de ningún objeto pero son menos poderosas y menos eficientes

## Archivos de texto

El trabajo con archivos en .Net está ligado al concepto de stream o flujo de datos, que consiste en tratar su contenido como una secuencia ordenada de datos. El concepto stream es aplicable también a otros tipos de almacenes de red o buffers en memoria.

La BCL proporciona las clases StreamReader y StreamWriter. Los objetos de estas clases facilitan la lectura y escritura de archivos de textos.

**StreamReader:** Para facilitar la lectura de flujos de texto StreamReader ofrece una familia de métodos que permiten leer sus caracteres de diferentes formas:

--> De uno en uno: El método int Read() devuelve el próximo carácter del flujo. Tras cada lectura la posición actual en el flujo se mueve un carácter hacia adelante.

--> Por líneas: El método string ReadLine() devuelve la siguiente línea del flujo (y avanza la posición en el flujo).

--> Por completo: string ReadToEnd(), que nos devuelve una cadena con todo el texto que hubiese desde la posición actual hasta el final (y avanza hasta el final del flujo)

**StreamWriter:** Ofrece los siguientes métodos

--> Escribir cadenas de texto: El método Write() escribe cualquier cadena de texto en el destino que tenga asociado. Pueden utilizarse formatos compuestos.

--> Escribir líneas de texto: El método WriteLine() funciona igual que Write() pero añade un indicador de fin de línea. Pueden utilizarse formatos compuestos

Para saber si es necesario liberar recursos Explícitamente cuando dejamos de usar un objeto de la BCL en nuestro código **hay que verificar si implementa la interfaz IDisposable**

```
public interface IDisposable
{
    void Dispose();
}
```

```
StreamReader sr = new StreamReader("fuente.txt");
StreamWriter sw = new StreamWriter("destino.txt");
string? linea;
while (!sr.EndOfStream)
{
    linea = sr.ReadLine();
    sw.WriteLine(linea);
}
sr.Close(); sw.Close();
```

El método close() libera los recursos de manera explícita, invocando un método Dispose()

IDisposable define un mecanismo determinista para liberar recursos no administrados y evita los problemas relacionados con el recolector de basura inherentes a los finalizadores.

Cuando se termina de usar un objeto que implementa IDisposable, se debe invocar el método Dispose() del objeto. Hay dos maneras de hacerlo:

- Mediante un bloque try/finally
- Mediante la instrucción using (no es la directiva using que venimos usando para hacer referencia a los espacios de nombres)

```
try
{
    using StreamReader sr = new StreamReader("fuente.txt");
    using StreamWriter sw = new StreamWriter("destino.txt");
    sw.WriteLine(sr.ReadToEnd());
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

# Principio de inversión de dependencias (DIP)

Y

## Patrón de Inyección de dependencias

Las aplicaciones web ASP.NET core hacen uso intensivo del principio de inversión de dependencias. Sin embargo este principio es muy poderoso y deberíamos aprovecharlo en cualquier tipo de aplicación.

El principio de inversión de dependencias (DIP) es uno de los 5 principios SOLID.

El principio Open/Closed, que vimos cuando introdujimos el tema del polimorfismo, también es uno de los principios SOLID.

Los principios SOLID facilitan el mantenimiento, extensión y reusabilidad del código.

Las clases de alto nivel no deberían depender de clases de menor nivel sino de abstracciones (interfaces o clases abstractas).

**Las clases de alto nivel son las que contienen la lógica de negocio, son las más importantes, establecen qué es y qué hace nuestra aplicación**

Para poder usar el patrón de inyección de dependencias lo que debemos lograr es que otra clase debe hacerse responsable de crear la instancia requerida e injectarla en la clase que queremos.

La inyección podría hacerse de varias maneras (por medio de método, propiedad o constructor). Nosotros utilizamos la inyección por constructor.

```
namespace CalculoSimple;
class Calculador(ILogger logger)
{
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```

En este ejemplo lo que hacemos es injectar en calculador la interfaz ILogger

## La clase que contiene a Main es de bajo nivel

La clase Program se considera una clase de muy bajo nivel. Es el punto de entrada inicial del sistema. Nada, salvo el sistema operativo, depende de ella. Por lo tanto se toleran las dependencias con otras clases de la aplicación sin afectar el principio de inversión de dependencias. En esta clase podemos establecer las configuraciones iniciales y luego entregar el control a módulos abstractos de alto nivel

El principio de inversión de dependencias favorece el acoplamiento débil en el código. Hace más fácil la modificación o reemplazo de un módulo por otro

Diseño fuertemente acoplado



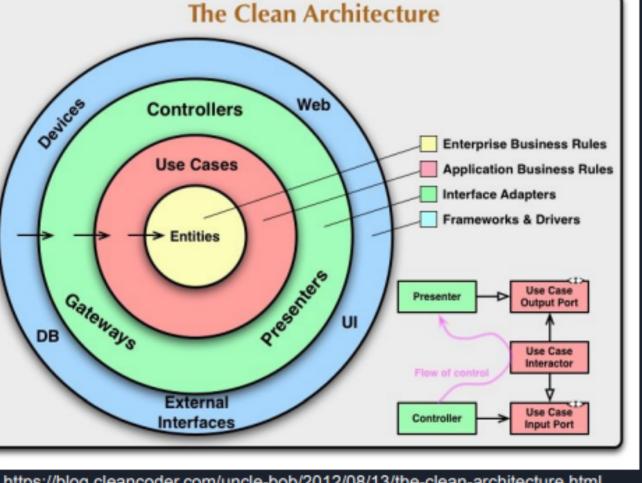
Diseño débilmente acoplado



Aplicar DIP hace que el código sea mantenible. Los programas pequeños, son inherentemente mantenibles, por ello aplicar DIP en ejemplos simples tiende a parecer una ingeniería excesiva. Cuanto mayor sea el tamaño del código, más visibles serán los beneficios de DIP

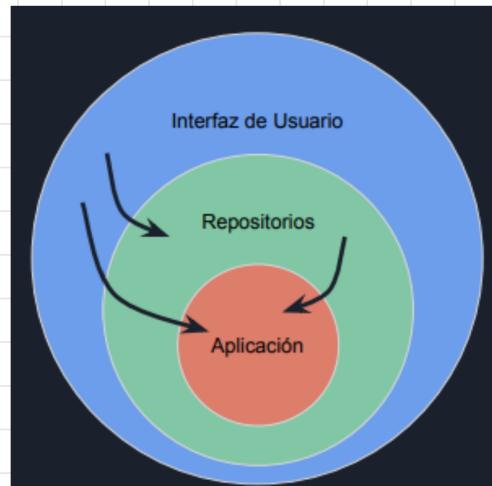
## Arquitectura Limpia

Se refiere a organizar el proyecto para que sea fácil de entender y cambiar a medida que el proyecto crece. Basada en la separación de responsabilidades o intereses. Para ello divide el software en capas.



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Para trabajar en este curso,  
proponemos una versión  
simplificada a esta.



Aplicación y Repositorios serán proyectos de biblioteca de clases.

Interfaz de usuario será un proyecto ejecutable (por ejemplo una aplicación de consola o Blazor)

Interfaz de usuario hará referencia a los proyectos Repositorios y Aplicación.

Repositorios (persistencia de datos, definiremos al menos un repositorio por cada entidad que debemos persistir) hará referencia a Aplicación.

Aplicación (lógica de las aplicaciones, entidades, casos de uso) no hará referencia a ningún otro proyecto.

**Nota:** Despues de esta explicación muestra como debería quedar resuelta la primera entrega aprox.

# Teoría 8

## Interfaces - Herencia

Las interfaces pueden heredar de múltiples interfaces

```
interface IInterface1 {  
    void Metodo1();  
}  
interface IInterface2 {  
    void Metodo2();  
}  
interface IInterface3: IInterface1, IInterface2 {  
    void Metodo3();  
}  
  
class A : IInterface3 {  
    ...  
}
```

La clase A debe implementar Metodo1(), Metodo2() y Metodo3()

Muy posiblemente los métodos de igual nombre pero de distintas interfaces, difieran semánticamente.  
Entonces, implementamos las interfaces explícitamente.



Cuando hay implementaciones explícitas de miembros de interfaz, la implementación a nivel de clase está permitida pero no es requerida.

Por lo tanto se tienen los siguientes 3 escenarios:

--> Una implementación a nivel de clase

--> Una implementación explícita de interface

--> Ambas, una implementación explícita de interface y una implementación a nivel de clase.

```
class A : IInterface1, IInterface2 {  
    ...  
    void IInterface1.Metodo() =>  
        Console.WriteLine("método de Interface1");  
    void IInterface2.Metodo() =>  
        Console.WriteLine("método de Interface2");  
    public void Metodo() =>  
        Console.WriteLine("método a nivel de la clase");  
    ...  
}  
  
A objA = new A();  
(objA as IInterface1).Metodo();  
(objA as IInterface2).Metodo();  
objA.Metodo();  
...
```

IMPORTANTE:  
La implementación explícita de un método de interface no lleva el modificador de acceso `public`

método de Interface1  
método de Interface2  
método a nivel de la clase

## Interfaces de la plataforma que se usan para la comparación

**Interface IComparable:** El método Sort de array funciona correctamente porque todos los elementos del vector (en este caso int) son comparables entre sí porque implementan la interface IComparable

```
var vector = new int[] { 27, 5, 100, -1, 3 };
Array.Sort(vector);
foreach (int i in vector)
{
    Console.WriteLine(i);
}
```

Ordenar un vector es muy simple utilizando el método estático Sort de la clase Array

Aunque no podemos modificar el método Sort() de array podemos hacer que funcione con nuestras clases enseñando a los objetos de estas clases a compararse entre sí implementando la interfaz IComparable.

-1  
3  
5  
27  
100

### Valores de retorno del método CompareTo

- ( <0 ) si this está antes que obj
- ( = 0 ) Si this ocupa la misma posición que obj
- ( > 0 ) Si this está después que obj

**Interface IComparer:** Si queremos otro criterio de orden, podemos utilizar una sobrecarga del método Array.Sort() que recibe también como argumento un objeto comparador que debe implementar la interfaz IComparer

## Interfaces de la plataforma que se utilizan para "enumerar".

**System.Collections.IEnumerable y System.Collections.IEnumerator**

Un tipo es enumerable si implementa la interface System.Collections.IEnumerable

Made with Goodnotes

```
namespace System.Collections
{
    public interface IEnumerable
    {
        // Returns an enumerator that
        // iterates through a collection.
        IEnumerator GetEnumerator();
    }
}
```

El método Getenumerator() devuelve un objeto de tipo interface, es decir de algún tipo que implemente la interfaz System.Collections.IEnumerator

Un enumerador es un objeto que puede devolver los elementos de una colección, uno por uno, en orden, según se solicite. Un enumerador "conoce" el orden de los elementos y realiza un seguimiento de dónde está en la secuencia. Luego devuelve el elemento actual cuando se solicita

```
namespace System.Collections
{
    public interface IEnumerator
    {
        // Gets the current element in the current position.
        object Current { get; }

        // Advances the enumerator to the next element
        // Returns true if the enumerator was successfully advanced
        bool MoveNext();

        // Sets the enumerator before the first element
        void Reset();
    }
}
```

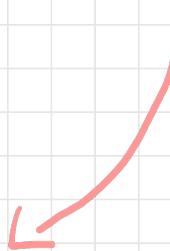
```
using System.Collections;

string[] vector = ["uno", "dos", "tres"];
IEnumerator e = vector.GetEnumerator();

while (e.MoveNext())
{
    Console.WriteLine(e.Current);
}
```

Invocar aquí e.Current provocaría una excepción InvalidOperationException. Lo mismo ocurriría después de e.Reset()  
**Tip:** Sólo invocar e.Current luego de obtener true con e.MoveNext()

### Recorriendo un enumerador



La instrucción foreach no necesita que la colección implemente la interfaz Ienumerable, sin embargo exige que exista un método con el nombre GetEnumerator() que devuelva un objeto que implemente la interfaz IEnumerator

## Iteradores

Los iteradores constituyen una forma mucho más simple de crear enumeradores y enumerables (el compilador lo hace por nosotros).

Utilizan la sentencia yield

- yield return: devuelve un elemento de una colección y mueve la posición al siguiente elemento
- yield break: detiene la iteración

Un bloque iterador es un bloque de código que contiene una o más sentencias yield. Puede contener múltiples sentencias yield return o yield break pero no se permiten sentencias return. El tipo de retorno de un bloque iterador debe declararse IEnumerator o IEnumerable.



**Nota:** Un enumerador generado con un iterador no implementa el método Reset()

Un iterador produce un enumerador, y no una lista de elementos. Este enumerador es invocado por la instrucción foreach. Esto permite iterar a través de grandes cantidades de datos sin leer todos los datos en memoria de una vez.

## Delegados

Tipo especial de clase cuyos objetos almacenan referencias a uno o más métodos de manera de poder ejecutar en cadena esos métodos. Permiten pasar métodos como parámetros a otros métodos. Proporcionan un mecanismo para implementar eventos.

Las variables que admiten métodos son de algún tipo delegado

Para definir un tipo de delegado, se usa una sintaxis similar a la definición de una firma de método. Solo hace falta agregar la palabra clave delegate a la definición.

**delegate int FuncionEntera(int n);**

El compilador genera una clase derivada de System.Delegate que coincide con la firma usada (en este caso, un método que devuelve un entero y tiene un argumento entero)

```
-----FuncionEntera.cs-----  
namespace Teoria8;  
delegate int FuncionEntera(int n);  
  
-----Auxiliar.cs-----  
namespace Teoria8;  
class Auxiliar  
{  
    public void Procesar()  
    {  
        FuncionEntera f;  
        f = SumaUno;  
        Console.WriteLine(f(10));  
        f = SumaDos;  
        Console.WriteLine(f(10));  
    }  
    int SumaUno(int n) => n + 1;  
    int SumaDos(int n) => n + 2;  
}
```

Se invoca  
SumaUno por  
medio de f

Se invoca  
SumaDos por  
medio de f

11  
12

También se pueden invocar los métodos en los delegados de forma explícita utilizando el método Invoke

Las variables de tipo delegado pueden asignarse directamente con el nombre del método o con su correspondiente constructor pasando el método como parámetro.

f = SumaUno; es equivalente a: f = new FuncionEntera(SumaUno);

```
...  
void Aplicar(int[] v, FuncionEntera f)  
{  
    for (int i = 0; i < v.Length; i++)  
    {  
        v[i] = f(v[i]);  
    }  
}  
void Imprimir(int[] v)  
{  
    foreach (int i in v)  
    {  
        Console.Write(i + " ");  
    }  
    Console.WriteLine();  
}
```

Made with Goodnotes

Recibe como  
parámetros un  
vector y una  
función en un  
delegado

Aplica la función f a  
cada uno de los  
elementos del vector v

```
namespace Teoria8;  
class Auxiliar  
{  
    public void Procesar()  
    {  
        int[] v = [11, 5, 90];  
        Aplicar(v, SumaDos);  
        Imprimir(v);  
        Aplicar(v, SumaUno);  
        Imprimir(v);  
    }  
    int SumaUno(int n) => n + 1;  
    int SumaDos(int n) => n + 2;  
    void Aplicar(int[] v, FuncionEntera
```

## Métodos anónimos

En ocasiones se definen métodos con la intención de ser invocados sólo por medio de una variable de tipo delegado.

Los métodos anónimos permiten prescindir del método con nombre definido por separado.

Un método anónimo es un método que se declara en línea, en el momento de crear una instancia de un delegado.

La sintaxis de un método anónimo incluye:

--> La palabra clave delegate

--> La lista de parámetros (si son necesarios)

--> El bloque de sentencias con la implementación del método.

----- delegate (parámetros) {implementación}; -----

Tiene la forma de un método cambiando su nombre por la palabra clave delegate y sin tipo de retorno. El tipo de retorno debe coincidir con el de la variable delegado a la que se asigne.

Los métodos anónimos pueden acceder a sus variables locales y a las definidas en el entorno que lo rodea (variables externas).

```
int externa = 7;
FuncionEntera f = delegate (int n)
{
    return n * 2 + externa;
};
Console.WriteLine(f(10));
```

## Expresiones lambda

Se puede transformar un método anónimo en una expresión lambda haciendo lo siguiente:

- Eliminar la palabra clave delegado.

- Colocar el operador lambda (`=>`) entre la lista de parámetros y el cuerpo del método anónimo.

```
f = delegate (int n) { return n * 2; };
f = (int n) => { return n * 2; };
```

Expresión  
lambda

Pero aún es posible otras simplificaciones estáticas:

--> si no existen parámetros ref, in, out, el tipo de los parámetros puede omitirse:

```
f = (n) => { return n * 2; };
```

--> si hay un único parámetro, pueden omitirse los paréntesis:

```
f = n => { return n * 2; };
```

--> si el bloque de instrucciones es sólo una expresión de retorno, puede reemplazarse todo el bloque por la expresión de retorno:

```
f = n => n * 2;
```

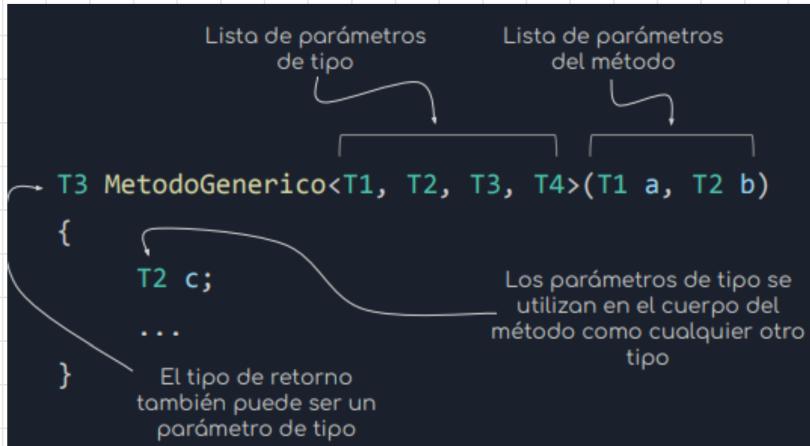
Nota: si el delegado no tiene parámetros se deben usar paréntesis vacíos:

```
linea = () => Console.WriteLine();
```

## Teoría 9

### Métodos genéricos

Los métodos genéricos permiten pasar los tipos como parámetros.



Si se pasan parámetros en la invocación a un método genérico, el compilador a veces puede inferir a partir de ellos los parámetros de tipo. Por lo tanto, el parámetro de tipo puede omitirse en la invocación.

```
void Swap<T>(ref T i, ref T j)
```

A partir del tipo de `i` y `j` pasados en la invocación, puede inferirse `T`

La solución que usa un método genérico es más eficiente:

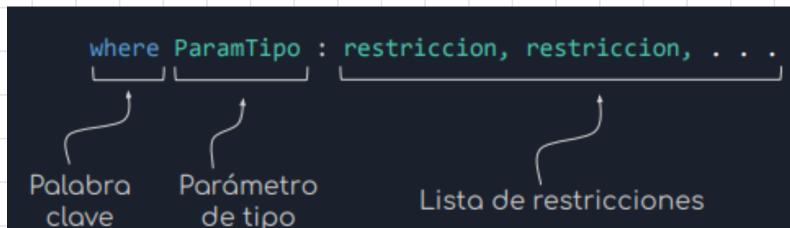
- No hay conversiones de tipo innecesarias, ni boxing ni unboxing.
- La seguridad de tipos es más fuerte.

## Restricciones sobre los parámetros de tipo - Cláusulas where

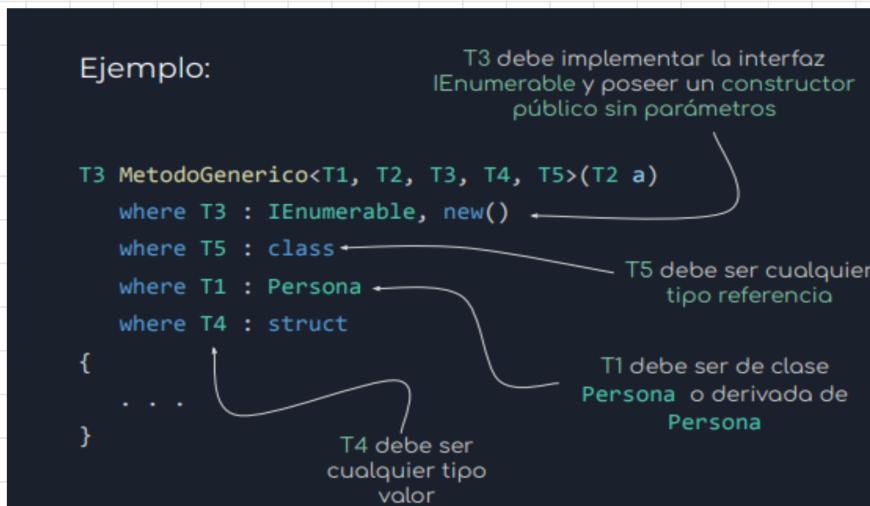
La restricciones se enumeran como cláusulas where.

--> Cada parámetro de tipo que tiene restricciones tiene su propia cláusula where

--> Si un parámetro tiene múltiples restricciones, se enumeran en la cláusula where, separadas por comas.



Si hay más de una cláusula where no se separan por comas ni ningún otro token. Se pueden enumerar en cualquier orden.



# Tipos genéricos

Además de los métodos ya vistos, C# provee cuatro categorías más de genéricos, todos ellos son tipos:

1. Clases
2. Estructuras
3. Interfaces
4. Delegados

Un tipo genérico no es un tipo real, sino una plantilla para un tipo real que se construye cuando se proporcionan los argumentos para los parámetros de tipo definidos

## Clases Genéricas

Las clases también admiten parámetros de tipo, estas clases se denominan clases genéricas.

Transformamos la clase `Par` en una clase genérica

```
class Par<T1, T2>
{
    public T1 A { get; private set; }
    public T2 B { get; private set; }
    public Par(T1 a, T2 b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Agregamos parámetros de tipo en la definición de la clase `Par`

Observar que el constructor lleva el nombre de la clase sin la lista de parámetros de tipo, es decir `Par(...)` no `Par<T1,T2>(...)`

```
Par<string, double>

class Par<T1, T2>
{
    public T1 A
    { get; private set; }
    public T2 B
    { get; private set; }
    public Par(T1 a, T2 b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Clase genérica

```
class Par<string, double>
{
    public string A
    { get; private set; }
    public double B
    { get; private set; }
    public Par(string a, double b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Clase construida

## Del tipo genérico al construido

Al indicar cuáles son los tipos reales (argumentos de tipo) que deben sustituir a los parámetros de tipo, el compilador JIT toma esos argumentos y crea el tipo real (que se llama tipo construido) del cual se podrán instanciar objetos

## Cláusulas where en clases genéricas

Valen los mismos considerando expuestos para el caso de métodos genéricos.

Las cláusulas where se colocan antes del cuerpo de la clase, ejemplo:

```
class ClaseGenerica<T1, T2, T3, T4>
    where T2 : IEnumerable
    where T3 : class
    where T4 : struct
{
    ...
}
```

## Colecciones genéricas

- `List<T>`: una de las más utilizadas, es la versión genérica de `ArrayList`
- `Dictionary< TKey, TValue >`: es una versión genérica de `Hashtable`
- `SortedDictionary< TKey, TValue >`: Idem al anterior pero ordenado según la clave
- `Queue<T>`: versión genérica de `Queue`
- `Stack<T>`: versión genérica de `Stack`
- `SortedSet<T>`: colección de elementos ordenados y sin duplicación
- `HashSet<T>`: conjunto de elementos sin duplicados sin orden en particular

La creación de instancias de clases genéricas con tipos específicos no duplica estas clases en el código IL. Sin embargo, cuando el compilador JIT compila las clases genéricas a código nativo, se crea una nueva clase para cada tipo de valor específico. Los tipos de referencia comparten la misma implementación de la misma clase nativa.

## Interfaces genéricas

Las interfaces genéricas permiten usar parámetros de tipo genérico en la declaración de sus miembros.

```
interface IRetornador<T>
{
    T Retornar(T valor);
}
```

Ejemplo de una interfaz genérica sencilla

Al establecer diferentes argumentos de tipo en una interfaz genérica se construyen distintas interfaces.

Ejemplo:

```
class Simple : IRetornador<int>, IRetornador<string> {...}
```

2 interfaces construidas

```
interface IRetornador<int>
{
    int Retornar(int valor);
}
```

```
interface IRetornador<string>
{
    string Retornar(string valor);
}
```

```
class Generica<T1, T2> : IRetornador<T2>
{
    public T2 Retornar(T2 valor)
    {
        return valor;
    }
    ...
}
```

Cuando se construya la clase Generica con tipos reales, por ejemplo Generica<char, int> se construye también la interfaz IRetornador<int>

```
Simple s = new Simple();
int i = s.Retornar(3);
string st = s.Retornar("holá");
Console.WriteLine(${st} {i});
```

```
class Simple : IRetornador<int>, IRetornador<string> {
    public int Retornar(int valor) {
        return valor;
    }
    public string Retornar(string valor) {
        return valor.ToUpper();
    }
}
```

HOLA 3

Observar que la clase Simple no es una clase genérica

Un parámetro de tipo de una clase genérica, puede usarse también como tipo de una interfaz genérica implementada por esa clase

Nota: No se puede implementar una interfaz con parámetro de tipo genérico y la misma interfaz construída con un tipo real.

.Net ofrece muchas interfaces genéricas para diferentes escenarios. Entre muchas otras están IComparer<T>, IComparable<T>, IEnumerator<T> e IEnumerable<T>.

A menudo existen versiones anteriores no genéricas de la misma interfaz basadas en el tipo object. Las respectivas versiones genéricas son preferibles porque evitan conversiones de tipos como boxing y unboxing.

## Extensión de métodos en interfaces genéricas

Extender métodos para una interfaz resulta de mucha utilidad ya que los métodos de extensión podrán invocarse en todos los tipos que implementan dicha interfaz

Made with Goodnotes

### Ejemplo

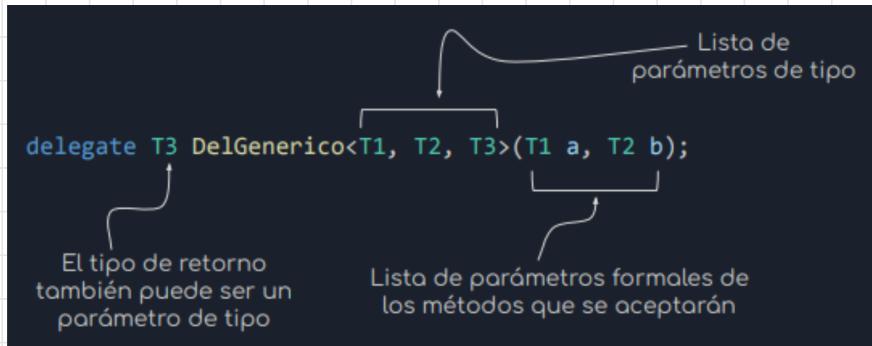
```
public static string EnUnaLinea<T>(this IEnumerable<T> secuencia)
{
    string resultado = "";
    foreach (T t in secuencia) resultado += $"{t} ";
    return resultado;
}
```

Luego podemos usar

```
int[] vector = [1, 2, 3, 4, 5];
List<string> lista = ["uno", "dos", "tres"];
string st1 = vector.EnUnaLinea(); // st1 ==> "1 2 3 4 5"
string st2 = lista.EnUnaLinea(); // st2 ==> "uno dos tres"
```

## Delegados genéricos

Los delegados genéricos se declaran como los delegados no genéricos pero utilizando parámetros en lugar de los tipos reales.



### **Delegado Action <T>**

.Net ofrece el delegado genérico `Action<T>` para métodos con tipo de retorno `void`.

- `Action<T1>` métodos con 1 parámetro
- `Action <T1, T2>` métodos con 2 parámetros

### **Delegado Func <TResult>**

.Net ofrece el delegado genérico `Func<TResult>` para métodos con tipo de retorno `TResult`

- `Func<TResult>` métodos sin parámetros
- `Func<T1, TResult>` métodos con 1 parámetro
- `Func<T1, T2, TResult>` métodos con 2 parámetros

### **Delegado Predicate <T>**

Presenta un método que determina si un objeto de tipo `T` cumple con determinados criterios

- `public delegate bool Predicate<T> (T obj);`

Existen métodos de la clase `List<T>` que reciben como parámetro un objeto `Predicate<T>`

- `-Find(Predicate <T>)` -----> `public T Find (Predicate<T> match);`  
→ `-FindAll(Predicate <T>)` -----> `List<T> FindAll (Predicate<T> match);`

## Delegado EventHandler<TEventArgs>

Para el manejo de eventos se provee el delegado genérico EventHandler<TEventArgs> que define un controlador que devuelve void y acepta dos parámetros, el primer parámetro es de tipo TEventArgs.

- public delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e);

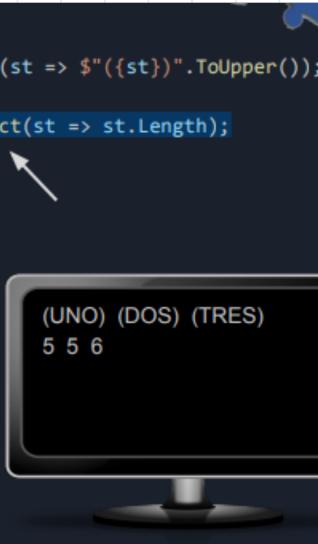
## Teoría 10

### LINQ

```
List<string> lista = ["uno", "dos", "tres"];
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());
Mostrar(secuencia);
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);
Mostrar(secuencia2);
```

Observar que secuencia2 es de un tipo distinto a secuencia (el método Select es un método genérico, se está haciendo inferencia de parámetros de tipos a partir del argumento, en este caso de tipo Func<string,int>), por lo tanto se está invocando a Select<string,int>

LINQ provee muchos otros métodos de extensión: select, where, Reverse, OrderBy, Sum, Average son algunos de ellos



Es muy común utilizar LINQ con inferencia de tipos (palabra clave var) para simplificar la escritura y lectura del código.

Otros métodos involucran a más de una secuencia, Concat, Union, Intersect y Zip son algunos

First, Last, Max, Min, All, Any son algunos métodos mas que brinda LINQ

A modo de definición: (no está en la teoría)

LINQ es una característica poderosa en .NET que permite realizar consultas sobre colecciones de datos de una manera similar a SQL. LINQ proporciona una sintaxis unificada y consistente para acceder a datos de diferentes fuentes, como bases de datos, colecciones en memoria, XML y más.

## Algunas de las ventajas de LINQ:

- Consistencia: Una única sintaxis para acceder a diferentes tipos de datos.
- Legibilidad: Código más fácil de leer y mantener.
- Tipado estático: Errores de tiempo de compilación en lugar de errores de tiempo de ejecución.
- Productividad: Menos código repetitivo y más centrado en la lógica de la aplicación

**Nota: En la teoría hay ejemplos de algunos métodos.**

## Persistencia de datos

Vamos a usar SQLite para persistir datos.

SQLite es una biblioteca que implementa un motor de base de datos SQL "en proceso", autónomo, sin servidor, de configuración cero.

SQLite es de código abierto y, por lo tanto, es gratuito para su uso para cualquier propósito

Nuestra aplicación de consola accederá a los datos mediante Entity Framework Core

Entity Framework Core es un ORM (object-relational mapper) que permite trabajar con una base de datos utilizando objetos .Net y ahorrando la escritura de mucho código de acceso a datos.

Con Entity Framework Core, el acceso a datos se realiza mediante un modelo. Un modelo se compone de clases de entidad y un objeto de contexto que representa una sesión con la base de datos. Este objeto de contexto permite consultar y guardar datos

Parte importante para el proyecto

Un objeto DbContext está diseñado para usarse durante una única unidad de trabajo, por lo que la duración de una instancia de un DbContext suele ser muy breve.

Es importante invocar al método Dispose() al terminar de usar el DbContext (en este caso lo realiza la instrucción using)

```
using Escuela;  
  
using (var context = new EscuelaContext())  
{  
    Console.WriteLine("-- Tabla Alumnos --");  
    foreach (var a in context.Alumnos)  
    {  
        Console.WriteLine($"{a.Id} {a.Nombre}");  
    }  
  
    Console.WriteLine("-- Tabla Exámenes --");  
    foreach (var ex in context.Exámenes)  
    {  
        Console.WriteLine($"{ex.Id} {ex.Materia} {ex.Nota}");  
    }  
}
```

Un  
para  
de t  
una

## Entity Framework Core - Code first

Si empezamos un proyecto nuevo, para el cual no existe una base de datos creada, podemos crearla fácilmente a partir del código C# que ya tenemos codificado. A esta estrategia se la conoce con el nombre de "Code First".

```
namespace Escuela;  
  
public class EscuelaSqlite  
{  
    public static void Inicializar()  
    {  
        using var context = new EscuelaContext();  
        if (context.Database.EnsureCreated())  
        {  
            Console.WriteLine("Se creó base de datos");  
        }  
    }  
}
```

Copiar el código del archivo 10\_RесурсыПоЛеории

Podemos utilizar una declaración using en lugar de la instrucción using

Si la base de datos no existe, se crea según el modelo definido en EscuelaContext y devuelve true

Y en la clase program lo que debemos poner es

EscuelaSqlite.Inicializar();

De esta forma se creará la base de datos automáticamente.

## Convenciones por defecto utilizadas:

- El nombre de las tablas se determinó a partir del nombre de las propiedades DbSet<> de EscuelaContext.
- La propiedad ID de las entidades se establecen como claves de las tablas.

- Si la propiedad ID de una entidad es entera, se establece como clave autoincremental.
- Sqlite soporta pocos tipos de datos, se utiliza un mapeo adecuado, por ejemplo las propiedades DateTime de .Net se establecen como campos TEXT en las tablas Sqlite.

## Existe un mecanismo más sofisticado para implementar "Code First" que se llama migraciones.

Con las migraciones es posible realizar modificaciones incrementales en la base de datos e incluso volver para atrás removiendo la última migración.

Para utilizarlas es necesario instalar las herramientas de Entity Framework necesarias.

Ejemplo de como agregar elementos a la base de datos:

```
namespace Escuela;

public class EscuelaSqlite
{
    public static void Inicializar()
    {
        using var context = new EscuelaContext();
        if (context.Database.EnsureCreated())
        {
            Console.WriteLine("Se creó base de datos");
            context.Add(new Alumno() { Nombre = "Juan", Email = "juan@gmail.com" });
            context.Add(new Alumno() { Nombre = "Ana" });
            context.Add(new Alumno() { Nombre = "Laura" });

            context.Add(new Examen() { AlumnoId = 2, Materia = "Inglés", Nota = 9,
                Fecha = DateTime.Parse("4/4/2022") });
            context.Add(new Examen() { AlumnoId = 1, Materia = "Inglés", Nota = 5,
                Fecha = DateTime.Parse("1/3/2019") });
            context.Add(new Examen() { AlumnoId = 1, Materia = "Álgebra", Nota = 10,
                Fecha = DateTime.Parse("24/5/2021") });

            context.SaveChanges();
        }
    }
}
```

Observar que podemos no especificar la propiedad `DbSet<>` donde agregar un `Alumno` o un `Examen`, porque existe una única propiedad `DbSet<>` por cada entidad.

Esta instrucción actualiza en la base de datos todos los cambios realizados (en este caso el alta de los alumnos y exámenes realizados)

Para realizar consultas usamos LINQ

```
using var context = new EscuelaContext();
var query = context.Alumnos.Join(context.Examenes,
    a => a.Id,
    e => e.AlumnoId,
    (a, e) => new
    {
        Alumno = a.Nombre,
        Materia = e.Materia,
        Nota = e.Nota
    });

```

Las propiedades de navegación, también facilitan el alta de información en las tablas relacionadas de la base de datos. Podemos agregar un nuevo alumno con la información de sus exámenes en una única operación SaveChanges()

```
using Microsoft.EntityFrameworkCore;
using Escuela;

using var db = new EscuelaContext();

Alumno nuevo = new Alumno()
{
    Nombre = "Andrés",
    Examenes = new List<Examen>() {
        new Examen(){Materia="Lengua",Nota=7,Fecha = DateTime.Parse("5/5/2022") },
        new Examen(){Materia="Matemática",Nota=6,Fecha = DateTime.Parse("6/5/2022") }
    }
};
db.Add(nuevo);
db.SaveChanges(); ← Se crea un alumno con su lista de exámenes y se salva de una sola vez

foreach (Alumno a in db.Alumnos.Include(a => a.Examenes))
{
    Console.WriteLine(a.Nombre);
    a.Examenes?.ToList()
        .ForEach(ex => Console.WriteLine($" - {ex.Materia} {ex.Nota}"));
}
```

## Integridad referencial en la base de datos:

Las propiedades de navegación también influyen en el comportamiento por defecto al momento de crear la base de datos.

Si creamos nuevamente la base de datos, eliminándola y volviendo a ejecutar EscuelaSqlite.Inicializar(); notaremos una diferencia importante.

Código SQL generado por Entity Framework para crear la tabla Examenes sin la propiedad de navegación:

```
CREATE TABLE "Examenes" (
    "Id" INTEGER NOT NULL CONSTRAINT "PK_Examenes" PRIMARY KEY AUTOINCREMENT,
    "AlumnoId" INTEGER NOT NULL,
    "Materia" TEXT NOT NULL,
    "Nota" REAL NOT NULL,
    "Fecha" TEXT NOT NULL
)
```

Con la propiedad de navegación:

```
CREATE TABLE "Examenes" (
    "Id" INTEGER NOT NULL CONSTRAINT "PK_Examenes" PRIMARY KEY AUTOINCREMENT,
    "AlumnoId" INTEGER NOT NULL,
    "Materia" TEXT NOT NULL,
    "Nota" REAL NOT NULL,
    "Fecha" TEXT NOT NULL,
    CONSTRAINT "Examenes_Alumnos_AlumnoId" FOREIGN KEY ("AlumnoId")
        REFERENCES "Alumnos" ("Id") ON DELETE CASCADE
)
```

```
CONSTRAINT "FK_Exámenes_Alumnos_AlumnoId" FOREIGN KEY ("AlumnoId")
    REFERENCES "Alumnos" ("Id") ON DELETE CASCADE
```

Esta instrucción establece una relación de integridad referencial entre la tabla Exámenes y la tabla Alumnos.

Los valores en la columna AlumnoId deben coincidir con valores existentes en la columna Id de la tabla Alumnos.

La opción ON DELETE CASCADE para indicar que si se elimina un registro en la tabla Alumnos, también se eliminarán automáticamente los registros asociados en la tabla Exámenes.

Esto es lo que se pide que hagamos  
en la segunda entrega del proyecto.

## Teoría 11

### Contenedor de Inyección de dependencias (DI-Container)

#### Configuración de las dependencias

Es conveniente agrupar la configuración de las dependencias en el código para facilitar futuros cambios en nuestra aplicación.

Idealmente para cambiar el comportamiento de la aplicación deberíamos:

- 1) Crear nuevas clases (dependencias) que implementen determinadas interfaces.
- 2) Configurar adecuadamente la elección de las dependencias que se utilizarán

Para concentrar en nuestro código la configuración de las dependencias podemos delegar en una única clase la creación de las instancias de todas las dependencias.

Como las dependencias pueden ser consideradas servicios, vamos a llamar a esa clase ProveedorServicios

## Contenedor de Inyección de dependencias

En lugar de la clase ProveedorServicios utilizada en el ejemplo anterior, usaremos un contenedor de inyección de dependencias.

Un contenedor de inyección de dependencias (DI Container) facilita configurar y obtener las dependencias que se usarán en la aplicación. También permite especificar si una dependencia debe usarse como singleton o debe crearse un nuevo objeto cada vez que se utilice.

Con "Singleton" nos referimos a una clase de la cual se va a instanciar un único objeto, por lo tanto la aplicación trabajará con la misma instancia en cualquier parte del código. Singleton es también un patrón de diseño .Net provee un contenedor de inyección de dependencias por medio de las clases ServiceCollection y ServiceProvider.

Para utilizar estas clases en una aplicación de consola es necesario instalar un paquete NuGet. Nuget es un administrador de paquetes gratuito y de código abierto diseñado para .Net.

comando:

```
dotnet add package Microsoft.Extensions.Hosting
```

Este es el nombre del paquete  
que se requiere instalar

Primero debemos agregar los paquetes Nuget y luego "invocarlos" a nuestro programa de esta forma

```
using DiContainer;
using Microsoft.Extensions.DependencyInjection;

var servicios = new ServiceCollection();
servicios.AddTransient<ILogger, LoggerConsola>();
servicios.AddTransient<IServicioX, ServicioX>();
var proveedor = servicios.BuildServiceProvider();

var servicioX = proveedor.GetService<IServicioX>();
servicioX?.Ejecutar();

var logger = proveedor.GetService<ILogger>();
logger?.Log("Fin del programa");
```

Agregar esta  
directiva using

La clase  
ProveedorServicios  
ya no es necesaria

Made with **Goodnotes**  
Se registran los servicios y se  
construye el proveedor

Copiar el código del archivo  
11\_RecursosParaLaTeoria

1

```
servicios.AddTransient<IServicioX, ServicioX>();  
servicios.AddTransient<ILogger, LoggerConsola>();
```

Se registran los servicio **IServicioX** y **ILogger** en la colección de servicios, indicando que cuando se requiera un **IServicioX** debe proveerse una nueva instancia de la clase **ServicioX** y cuando se requiera un **ILogger** debe proveerse una nueva instancia de la clase **LoggerConsola**



Así se utiliza en el proyecto, con todos los servicios y repositorios.

Para que el contenedor pueda proveer los servicios requeridos se necesita:

- 1) Haber registrado el servicio y todas sus dependencias en el contenedor.
- 2) Utilizar en todos los casos inyección por medio del constructor

### Tiempo de vida de los servicios en un contenedor

- **servicios.AddTransient<ILogger, LoggerConsola>();**

Registra el servicio **LoggerConsola** como transitorio. El proveedor devolverá un nuevo objeto cada vez que se lo requiera.

- **servicios.AddSingleton<ILogger,LoggerConsola>();**

Registra el servicio **LoggerConsola** como singleton. El proveedor devolverá siempre el mismo objeto cada vez que se lo requiera

Los servicios también se pueden registrar dentro de un scope (alcance o ámbito). Resulta útil en las aplicaciones web ASP.NET Core.

- **servicios.AddScoped<IServicioA, ServicioA>();**

Se devuelve la misma instancia dentro del mismo ámbito. Para una aplicación Blazor Server se crea un ámbito por cada conexión SignalR, las instancias se compartirán entre páginas y componentes para un mismo usuario, pero no entre diferentes usuarios y no entre diferentes pestañas del mismo explorador.

## Aplicaciones Web con ASP.NET Core Blazor

Blazor es un framework de interfaz de usuario para .Net, es parte de ASP.NET Core

Razor es un formato para generar contenido basado en texto como HTML. Los archivos Razor tienen una extensión de archivo cshtml o razor y contienen una combinación de código C# junto con HTML

Una aplicación Blazor Server se implementa en un servidor web. El servidor mantiene con el navegador del usuario un canal de comunicación bidireccional SignalR. Las acciones de los usuarios sobre la aplicación se transmiten por esta conexión SignalR al servidor y, si es necesario actualizar la interfaz de usuario, el framework de Blazor Server envía en tiempo real al navegador los cambios para que se apliquen a la interfaz de usuario

En una aplicación Blazor WebAssembly, las DLL de la aplicación se transmiten al navegador del usuario y se ejecutan sobre una versión de .NET optimizada para el entorno de ejecución WebAssembly del navegador.

Se desplaza todo el procesamiento de la aplicación a la máquina del usuario. Para obtener datos o interactuar con otros servicios, la aplicación puede usar tecnologías web estándar para comunicarse con servicios HTTP.

### Componentes

Las aplicaciones Blazor se basan en componentes. Un componente es un elemento de la interfaz de usuario como una página, un cuadro de diálogo o un formulario de entrada de datos. Utilizan sintaxis Razor (C# y HTML) y se escriben en archivos con extensión .razor. Los componentes se compilan en clases .Net, se pueden anidar y reutilizar. Pueden ser "ruteables" (directiva @page)

Por convención en la carpeta Components/Pages se colocan los componentes "ruteables"

## Expresiones implícitas y explícitas en Razor

Las expresiones implícitas Razor comienzan por @ seguida de código de C#. Generalmente no admiten espacios y no se indica dónde terminan, se trata de expresiones simples, por ejemplo `@nombre.ToUpper()`

En algunas expresiones es necesario indicar cuál es el comienzo y fin de la misma, se llaman expresiones explícitas y se denotan entre paréntesis, por ejemplo: `@(5*2)`

```
---- Hola.razor -----
@page "/hola"
@rendermode InteractiveServer
<button @onclick="Cambiar">Mostrar / Ocultar</button>
@if (EsVisible)
{
    <h1>Hola @nombre.ToUpper()</h1>
}
@code {
    string nombre = "Juan";
    bool EsVisible = true;
    void Cambiar()
    {
        EsVisible = !EsVisible;
    }
}
```

Copiar el código del archivo  
11\_RecursosParaLaTeoria.txt

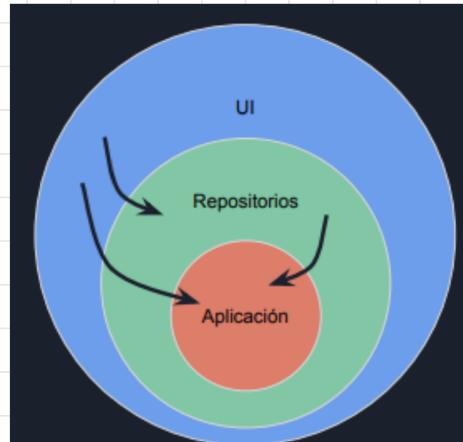
Es necesario si queremos que el componente sea interactivo

## Teoría 12

### Repaso para el proyecto final

En cuanto a la arquitectura limpia con la que venimos trabajando:

- Aplicación y Repositorios serán proyectos de biblioteca de clases.
- UI será un proyecto ejecutable (por ej. una aplicación de consola o Blazor)



- >UI hará referencia a los proyectos Repositorios y Aplicación.
- >Repositorios hará referencia a Aplicación
- >Aplicación no hará referencia a ningún otro proyecto

La aplicación Blazor que usaremos como interfaz de usuario viene con un DI container integrado. Registraremos nuestros servicios en la clase program por medio de builder.Services que devuelve un IServiceCollection

```
using AL.UI.Components;

//agregamos estas directivas using
using AL.Repositorios;
using AL.Aplicacion.UseCases;
using AL.Aplicacion.Interfaces;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();

//agregamos estos servicios al contenedor DI
builder.Services.AddTransient<AgregarClienteUseCase>();
builder.Services.AddTransient<ListarClientesUseCase>();
builder.Services.AddTransient<EliminarClienteUseCase>();
builder.Services.AddTransient<ModificarClienteUseCase>();
builder.Services.AddTransient<ObtenerClienteUseCase>();
builder.Services.AddScoped<IRepositorioCliente, RepositorioClienteMock>();

var app = builder.Build();
. . .
```

Copiar el código del archivo  
12\_RecursosParaLaTeoría

No hay más material teórico en esta clase, el resto es una muestra de lo que debemos hacer en la entrega final.