



.NET

Teoría 5

Miembros estáticos



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.
- Para declararlos se utiliza el modificador **static**



Miembros estáticos



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria5`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar la clase Cuenta en su propio archivo fuente Cuenta.cs



Método estático

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

Métodos estáticos

La sintaxis para acceder a un método, y en general a cualquier miembro estático de una clase es:

`<NombreClase>.<Miembro>`

por ejemplo:

`Cuenta.ImprimirResumen()`





Modificar Program.cs y ejecutar

```
using Teoria5;  
Cuenta.ImprimirResumen();
```





Modificar Program.cs y ejecutar

```
using Teoria5;  
Cuenta.ImprimirResumen();
```

No es necesario
instanciar
ningún objeto
de tipo Cuenta



No hay datos



Modificar Program.cs agregando las líneas resaltadas y verificar el mensaje del compilador

```
using Teoria5;  
Cuenta.ImprimirResumen();  
Cuenta c1 = new Cuenta();  
c1.ImprimirResumen();
```





Modificar Program.cs agregando las líneas resaltadas y verificar el mensaje del compilador

```
using Teoria5;  
Cuenta.ImprimirResumen();  
Cuenta c1 = new Cuenta();  
c1.ImprimirResumen();
```

No se puede obtener acceso al miembro 'Cuenta.ImprimirResumen()' con una referencia de instancia; califíquelo con un nombre de tipo en su lugar



Campos estáticos

- Un **campo estático** de una clase es una variable accesible a través de la clase en la 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.



Modificar la clase Cuenta

```
namespace Teoria5;  
class Cuenta  
{
```

```
    public int Monto;
```

```
    public static int Total;
```

```
    public static void ImprimirResumen()
```

```
    => Console.WriteLine($"Total = {Total}");
```

```
}
```

Campo
estático





Modificar Program.cs y ejecutar

```
using Teoria5;  
Cuenta c1 = new Cuenta();  
Cuenta c2 = new Cuenta();  
c1.Monto = 20;  
Cuenta.Total += c1.Monto;  
c2.Monto = 30;  
Cuenta.Total += c2.Monto;  
Cuenta.ImprimirResumen();
```



Miembros estáticos - Campos estáticos

----- Program.cs -----

```
Cuenta c1 = new Cuenta();  
Cuenta c2 = new Cuenta();  
c1.Monto = 20;  
Cuenta.Total += c1.Monto;  
c2.Monto = 30;  
Cuenta.Total += c2.Monto;  
Cuenta.ImprimirResumen();
```

----- Cuenta.cs -----

```
class Cuenta  
{  
    public int Monto;  
    public static int Total;  
    public static void ImprimirResumen()  
        => Console.WriteLine($"Total = {Total}");  
}
```

Existe una única variable **Cuenta.Total** en donde acumulamos los montos de las instancias c1 y c2



Total = 50

Memoria Heap

Cuenta

Total: 50

ImprimirResumen()

c1

Monto: 20

c2

Monto: 30

Podemos pensar a una clase como un objeto único, independiente de sus instancias, con sus propios campos y métodos (los estáticos). Además una clase no requiere ser creada explícitamente con una instrucción.



Modificar el método estático ImprimirResumen() y verificar el mensaje del compilador

```
namespace Teoria5;
class Cuenta
{
    public int Monto;
    public static int Total;
    public static void ImprimirResumen()
        => Console.WriteLine($"Monto = {Monto}");
}
```



Miembros estáticos - Campos estáticos

```
class Cuenta
{
    public int Monto;
    public static int Total;
    public static void ImprimirResumen()
        => Console.WriteLine($"Monto = {Monto}");
}
```

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

Se requiere una referencia de objeto
para el campo, método o propiedad
'Cuenta.Monto' no estáticos

Miembros estáticos - Campos estáticos

```
class Cuenta
{
    public int Monto;
    public static int Total;
    public static void ImprimirResumen()
        => Console.WriteLine($"Monto = {Monto}");
}
```

La única variable
accesible desde
`ImprimirResumen` es
`Total`

Cuenta

Total: 50

Heap

`ImprimirResumen()`



----- Program.cs -----

```
Cuenta c1 = new Cuenta();  
c1.Monto = 20;  
Cuenta.Total = 0;  
c1.Total += 25;  
Cuenta.Monto += 5;
```

----- Cuenta.cs -----

```
class Cuenta  
{  
    public int Monto;  
    public static int Total;  
    public static void ImprimirResumen()  
        => Console.WriteLine($"Total = {Total}");  
}
```

Para pensar
¿Qué sentencias
son incorrectas?

Miembros estáticos - Campos estáticos

----- Program.cs -----

```
Cuenta c1 = new Cuenta();  
c1.Monto = 20;  
Cuenta.Total = 0;
```

```
x c1.Total += 25;
```

```
x Cuenta.Monto += 5;
```

Total es un campo estático, no se puede acceder a través de una instancia

Monto es un campo de instancia no puede accederse a través de una clase

----- Cuenta.cs -----

```
class Cuenta  
{  
    public int Monto;  
    public static int Total;  
    public static void ImprimirResumen()  
        => Console.WriteLine($"Total = {Total}");  
}
```

Convenciones de nomenclatura para campos privados

El equipo de **Microsoft** no tiene convenciones estrictas al respecto, sin embargo, el equipo de **.NET Core** adoptó usar el prefijo **_** (guión bajo) para campos privados y el prefijo **s_** para campos privados estáticos



Miembros estáticos - Campos estáticos

----- Program.cs -----

```
Cuenta c1 = new Cuenta();  
Cuenta c2 = new Cuenta();  
c1.Depositar(100);  
c2.Depositar(200);  
c1.Depositar(300);  
Cuenta.ImprimirResumen();  
c1.Imprimir();  
c2.Imprimir();
```

----- Cuenta.cs -----

```
class Cuenta {  
    private int _monto;  
    private static int s_total;  
  
    public static void ImprimirResumen()  
        => Console.WriteLine($"Total = {Cuenta.s_total}");  
  
    public void Depositar(int monto) {  
        _monto += monto;  
        Cuenta.s_total += monto;  
    }  
  
    public void Imprimir() => Console.WriteLine(_monto);  
}
```

Para pensar
¿Qué hace este
programa?

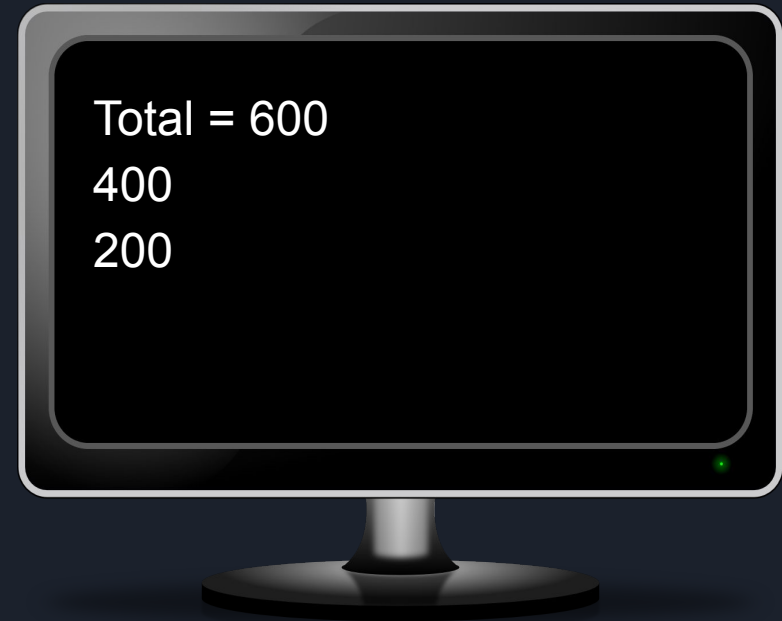
Miembros estáticos - Campos estáticos

----- Program.cs -----

```
Cuenta c1 = new Cuenta();  
Cuenta c2 = new Cuenta();  
c1.Depositar(100);  
c2.Depositar(200);  
c1.Depositar(300);  
Cuenta.ImprimirResumen();  
c1.Imprimir();  
c2.Imprimir();
```

----- Cuenta.cs -----

```
class Cuenta {  
    private int _monto;  
    private static int s_total;  
  
    public static void ImprimirResumen()  
        => Console.WriteLine($"Total = {Cuenta.s_total}");  
  
    public void Depositar(int monto) {  
        _monto += monto;  
        Cuenta.s_total += monto;  
    }  
  
    public void Imprimir() => Console.WriteLine(_monto);  
}
```



Heap

Cuenta

s_total: 600

ImprimirResumen()

c1

_monto: 400

Depositar(int)

c2

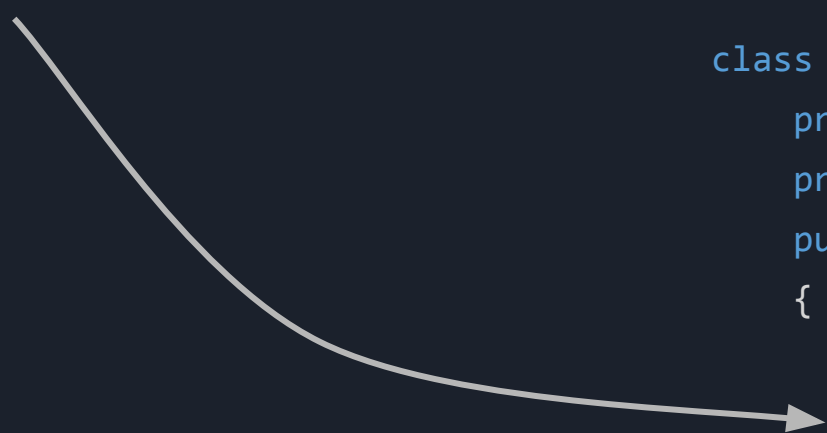
_monto: 200

Depositar(int)


```
class Cuenta {  
    private int _monto;  
    private static int s_total;  
    public void Depositar(int monto)  
    {  
        _monto += monto;  
        Cuenta.s_total += monto;  
    }  
    ...  
}
```

Los miembros
estáticos pueden
accederse desde la
propia clase sin
anteponer el nombre
de la misma.

```
class Cuenta {  
    private int _monto;  
    private static int s_total;  
    public void Depositar(int monto)  
    {  
        _monto += monto;  
        s_total += monto;  
    }  
    ...  
}
```



```
public void Depositar(int monto)
{
    _monto += monto;
    s_total += monto;
}
```

¿ Los campos `_monto` y `s_total` serán
estáticos o de instancia ?

La convención de nomenclatura
adoptada por el equipo de **.NET Core**
ayuda a entenderlo



```
public void Depositar(int monto)
{
    this.monto += monto;
    Cuenta.total += monto;
}
```

Así también es claro. Usando `this` y el nombre de la clase no se hace necesaria la utilización de prefijos en los nombres de los campos.



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



Constructores estáticos

- El **runtime de .NET** no garantiza cuándo 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

Constructores estáticos

- En C #, el constructor estático generalmente se ejecuta inmediatamente antes de la primera llamada a cualquier miembro de la clase.
- Es posible tener un constructor estático y un constructor de instancia ambos sin parámetros definidos en la misma clase.
- Si a los campos estáticos se les han dado valores predeterminados, estos se asignan antes de que se llame al constructor estático

Ejemplo

```
class ClaseA
{
    private static double valorFinal;
    private static double valorInicial = 15;
    static ClaseA() ← Constructor
                    ← estático
    {
        ClaseA.valorFinal = ClaseA.valorInicial * 2;
    }
    private double _valor;
    public ClaseA() ← Constructor
                  ← de instancia
    {
        _valor = (ClaseA.valorInicial + ClaseA.valorFinal) / 2;
    }
}
```

No hay conflicto
aunque la firma
de los
constructores
sea la misma.

Orden de ejecución provocado por la primera instrucción
`ClaseA a = new ClaseA();` en nuestro código


```
class ClaseA
{
    private static double valorFinal;
    private static double valorInicial = 15;
    static ClaseA()
    {
        ClaseA.valorFinal = ClaseA.valorInicial * 2;
    }
    private double _valor;
    public ClaseA()
    {
        _valor = (ClaseA.valorInicial + ClaseA.valorFinal) / 2;
    }
}
```


Clases estáticas

- Las clases estáticas llevan el modificador `static` en su declaración
- Sólo pueden poseer miembros estáticos
- No es posible instanciar objetos de una clase estática.
- A menudo agrupan un conjunto de utilidades y datos relacionados (*utility class*)
- Ejemplos de clases estáticas: `Console`, `File`, `Directory`, `Math`, etc.

Clases estáticas

Ejemplo



```
static class FechaActual
{
```

El compilador evita
cualquier intento de
instanciar un objeto
FechaActual

```
    public static void ImprimirHora() =>
        Console.WriteLine($"{DateTime.Now:hh:mm:ss}");

    public static void ImprimirFecha() =>
        Console.WriteLine($"{DateTime.Today:dd/MM/yyyy}");
}
```

Clases estáticas Ejemplo

```
----- Program.cs -----  
FechaActual.ImprimirFecha();  
FechaActual.ImprimirHora();
```

```
----- FechaActual.cs -----  
static class FechaActual  
{  
    public static void ImprimirHora() =>  
        Console.WriteLine($"{DateTime.Now:hh:mm:ss}");  
  
    public static void ImprimirFecha() =>  
        Console.WriteLine($"{DateTime.Today:dd/MM/yyyy}");  
}
```



Clases estáticas *Utility Classes*

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`.
- Deben `inicializarse` cuando se declaran.
- Son siempre `implícitamente estáticas`.
- Sin embargo `no se usa el modificador static`.

----- Program.cs -----

```
Console.WriteLine(A.PI);  
A a = new A();  
Console.WriteLine(a.PI);
```

----- A.cs -----

```
class A  
{  
    public const double PI = 3.1416;  
}
```

Para pensar
¿Dónde se
produce error de
compilación?



----- Program.cs -----

```
Console.WriteLine(A.PI);
```

```
A a = new A();
```

```
x Console.WriteLine(a.PI);
```

----- A.cs -----

```
class A
```

```
{
```

```
    public const double PI = 3.1416;
```

```
}
```

Ok

Error de compilación: No se puede obtener acceso al miembro 'A.PI' con una referencia de instancia; califíquelo con un nombre de tipo en su lugar

Campos constantes (`const`)

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`

IMPORTANTE

Las constantes puede
definirse únicamente de tipos
integrados numéricos, char,
bool o string

Nota: En realidad se pueden utilizar cualquier
otro tipo de referencia pero sólo se les puede
asignar el valor null



```
class A
{
    const double doblePI = PI * 2;
    const string saludoTotal = saludo + " Mundo!";
    const string saludoDos = "Hola" + 2;
    static double tres = 3.0;
    const double triplePI = PI * tres;
    const DateTime dia = DateTime.Now;
    const double PI = 3.1415;
    const string saludo = "Hola";
}
```

Para pensar
¿Dónde se
produce error de
compilación?



```
class A
```

```
{
```

```
    const double doblePI = PI * 2;
```

```
    const string saludoTotal = saludo + " Mundo!";
```

```
x const string saludoDos = "Hola" + 2;
```

```
    static double tres = 3.0;
```

```
x const double triplePI = PI * tres;
```


```
x const DateTime dia = DateTime.Now;
```

```
    const double PI = 3.1415;
```

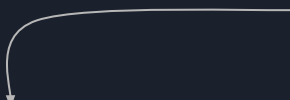
```
    const string saludo = "Hola";
```

```
}
```

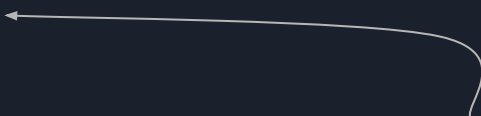
El compilador no sabe evaluar esta expresión



El compilador no puede acceder a una variable



DateTime no es un tipo que el compilador pueda inicializar



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*
- Sólo pueden asignarse en su *declaración* o dentro de un *constructor* (los estáticos cuando se declaran o en el constructor estático)
- 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

Campos *readonly*

```
class A
{
    const double doblePI = PI * 2;

    const string saludoTotal = saludo + " Mundo!";

    readonly string saludoDos = "Hola" + 2;

    static double tres = 3.0;

    static readonly double triplePI = PI * tres;

    readonly DateTime dia = DateTime.Now;

    public const double PI = 3.1416;

    const string saludo = "Hola";
}
```

Ok

Ok

Ok

- Un *campo readonly* se puede asignar varias veces, siempre que sea dentro de un constructor

Campos *readonly*

```
class ClaseA
```

```
{
```

```
    readonly DateTime fecha = DateTime.Now;
```

Ok

```
    public ClaseA(DateTime dt)
```

```
    {
```

```
        fecha = dt;
```

Ok

```
    }
```

```
    public ClaseA() : this(DateTime.Today.AddYears(1))
```

```
    {
```

```
        fecha = new DateTime(2000, 1, 1);
```

Ok

```
        fecha = fecha.AddDays(40);
```

```
    }
```

```
    public void Procesar()
```

```
    {
```

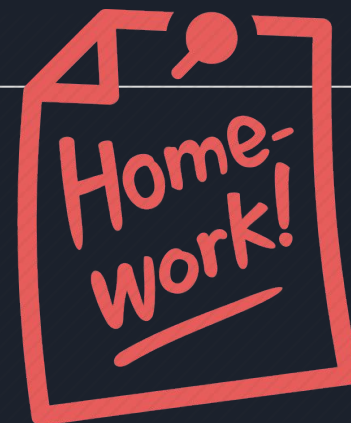
```
        x fecha = fecha.AddDays(10);
```

Ok

```
    }
```

Error de compilación

Campos *readonly* Para responder en casa



¿Hay algún error en esta clase?

```
class Coleccion
{
    private readonly List<object> lista = new List<object>();
    public void Agregar(object obj)
    {
        lista.Add(obj);
    }
    . . .
}
```

Encapsulamiento

Acceso a miembros privados



El rol del 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**.



Implementar la clase Cuadrado en su propio archivo fuente Cuadrado.cs

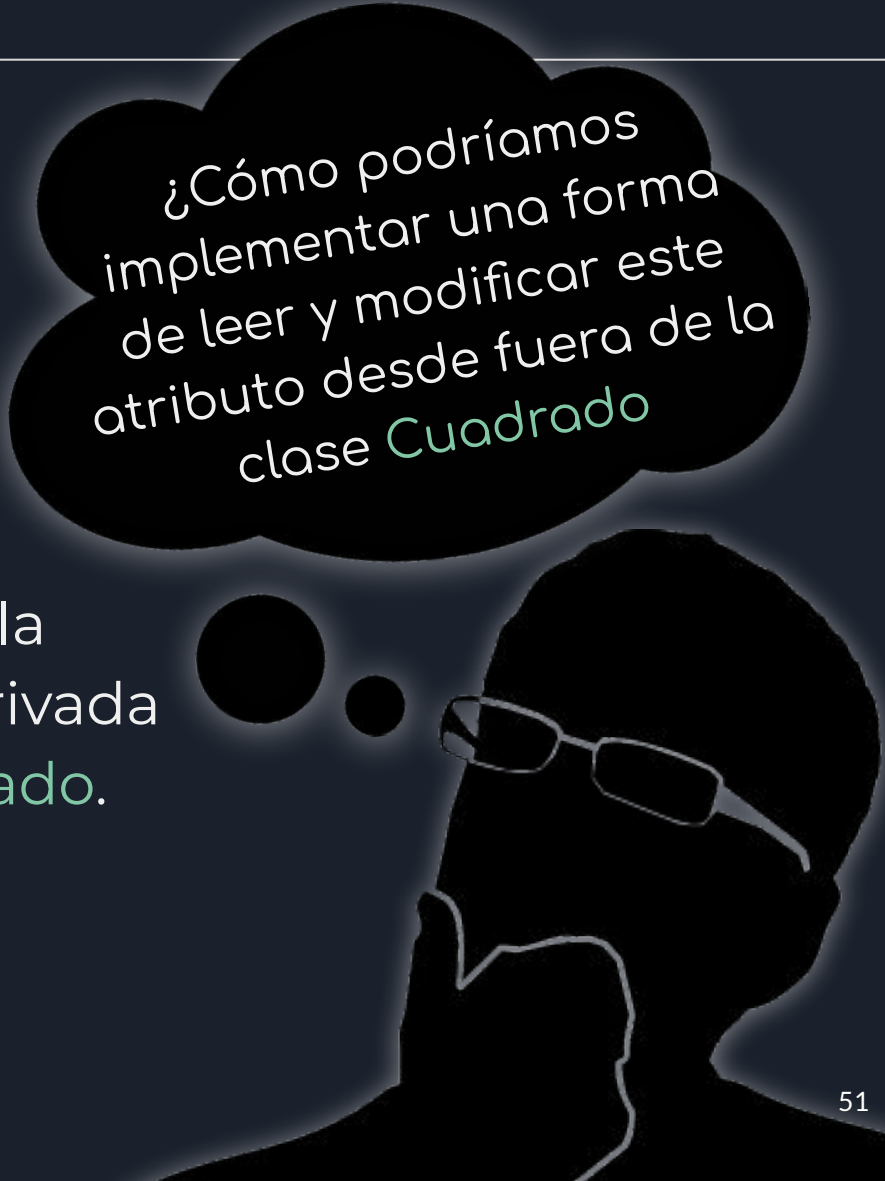
```
namespace Teoria5;  
class Cuadrado  
{  
    private double _lado;  
}
```



¿Cómo acceder a campos privados?

```
class Cuadrado  
{  
    private double _lado;  
}
```

El campo `_lado` constituye la representación interna y privada del atributo `lado del cuadrado`.



¿Cómo podríamos implementar una forma de leer y modificar este atributo desde fuera de la clase `Cuadrado`



Implementar los métodos getters y setters para la clase Cuadrado. Agregar otro método para obtener el área del cuadrado

setter

```
namespace Teoria5;
```

```
class Cuadrado
```

```
{
```

```
    private double _lado;
```

```
    public void SetLado(double value) => _lado = value;
```

```
    public double GetLado() => _lado;
```

```
    public double GetArea() => _lado * _lado;
```

```
}
```

getter





Codificar Program.cs y ejecutar

```
using Teoria5;
Cuadrado c = new Cuadrado();
c.SetLado(2.5);
Console.WriteLine($"Lado: {c.GetLado()} área: {c.GetArea()}");
```

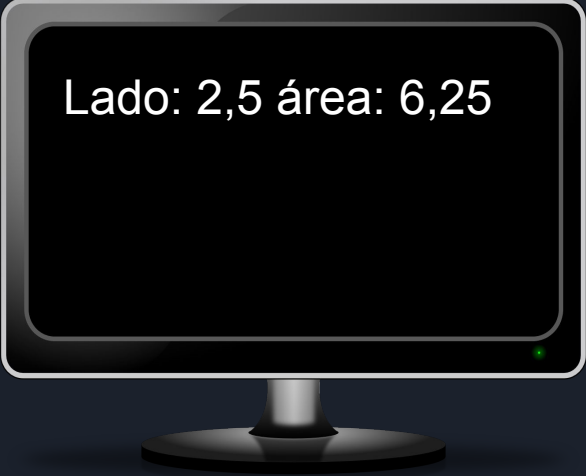


----- Program.cs -----

```
Cuadrado c = new Cuadrado();  
c.SetLado(2.5);  
Console.WriteLine($"Lado: {c.GetLado()} área: {c.GetArea()}");
```

----- Cuadrado.cs -----

```
class Cuadrado  
{  
    private double _lado;  
    public void SetLado(double value) => _lado = value;  
    public double GetLado() => _lado;  
    public double GetArea() => _lado * _lado;  
}
```



Lado: 2,5 área: 6,25

Cómo acceder a campos privados

Sin embargo, usar *getters* y *setters* como acabamos de hacer no es lo indicado en la plataforma .NET

Se deben usar
"Propiedades"



Propiedades

Propiedades

- Una propiedad integra los 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** (*accessors* en inglés)

Propiedades - Sintaxis

```
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

Nombre de la propiedad

Se debe devolver un objeto del tipo de la propiedad

Se recibe un valor en un parámetro implícito llamado `value` (mismo tipo que la propiedad)

Propiedades - Ejemplo

Código que utiliza la clase Cuadrado

...

```
Cuadrado c = new Cuadrado();
```

```
c.Lado = 2.5;
```

```
double lado = c.Lado;
```

...

Clase Cuadrado

```
public double Lado
```

```
{
```

```
    set
```

```
    {
```

```
        _lado = value;
```

```
    }
```

```
    get
```

```
    {
```

```
        return _lado;
```

```
    }
```

```
}
```

value = 2.5

Propiedades

- 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 el uso de propiedades de sólo escritura. Si la intención es desencadenar algún efecto secundario cuando se asigna el valor, es preferible usar un método en lugar de una propiedad.



Modificar la clase Cuadrado

```
namespace Teoria5;
class Cuadrado {
    private double _lado;

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

    public double Area {
        get {
            return _lado * _lado;
        }
    }
}
```

Propiedad de
lectura / escritura

Propiedad de sólo
lectura

Código en el archivo
05_RecursosParaLaTeoria.txt



Modificar Program.cs

```
using Teoria5;  
Cuadrado c = new Cuadrado();  
c.Lado = 2.5;  
Console.WriteLine("Lado: {c.Lado} área: {c.Area}");
```

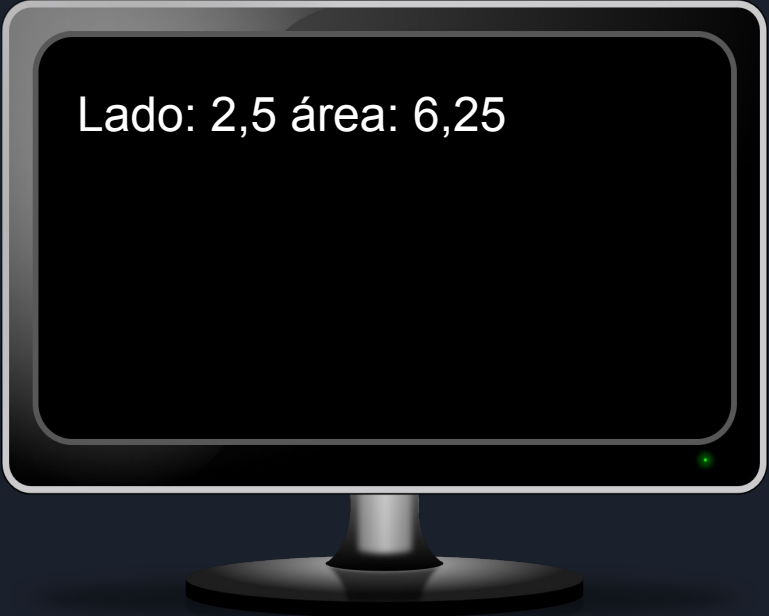


----- Program.cs -----

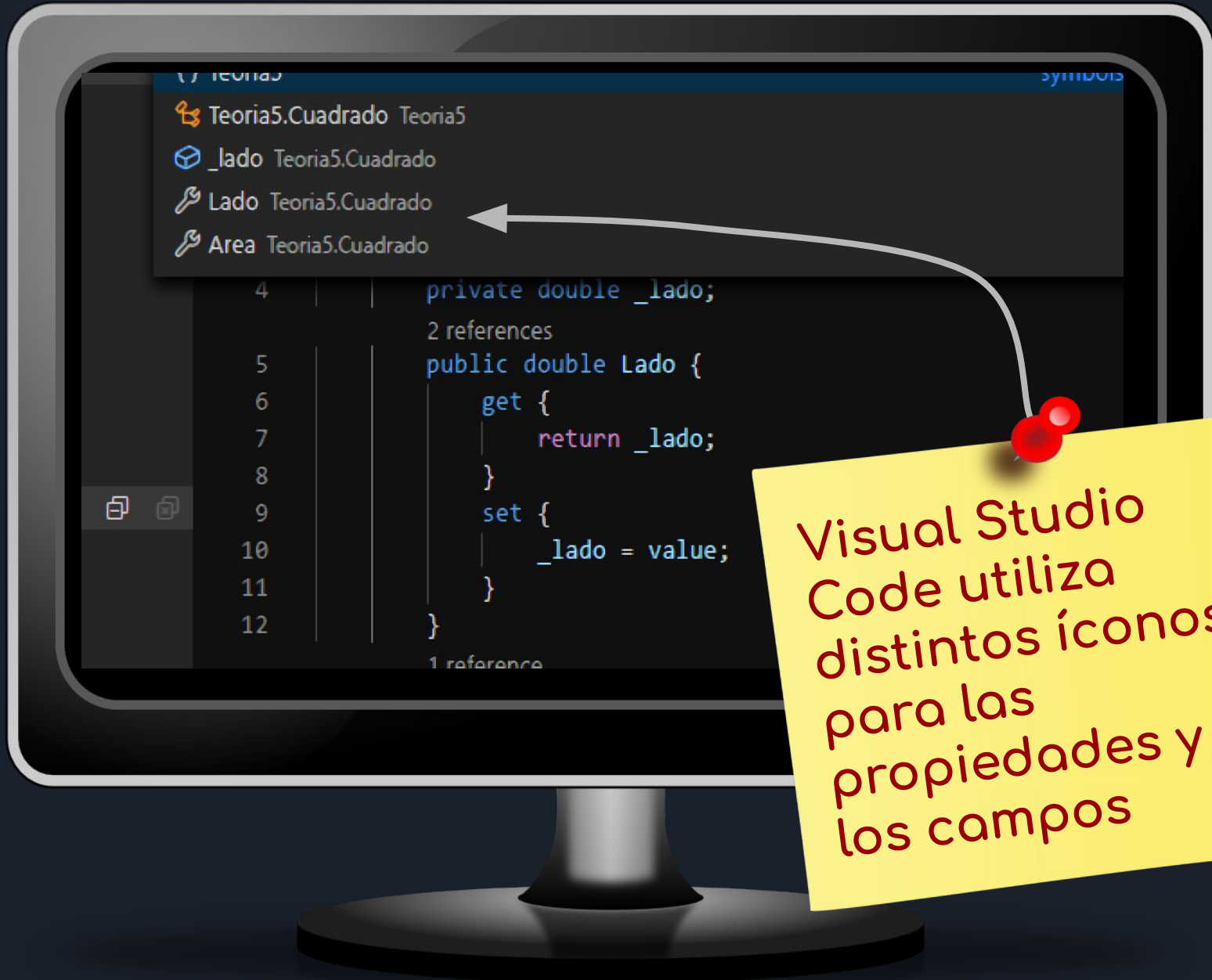
```
Cuadrado c = new Cuadrado();  
c.Lado = 2.5;  
Console.WriteLine("Lado: {c.Lado} área: {c.Area}");
```

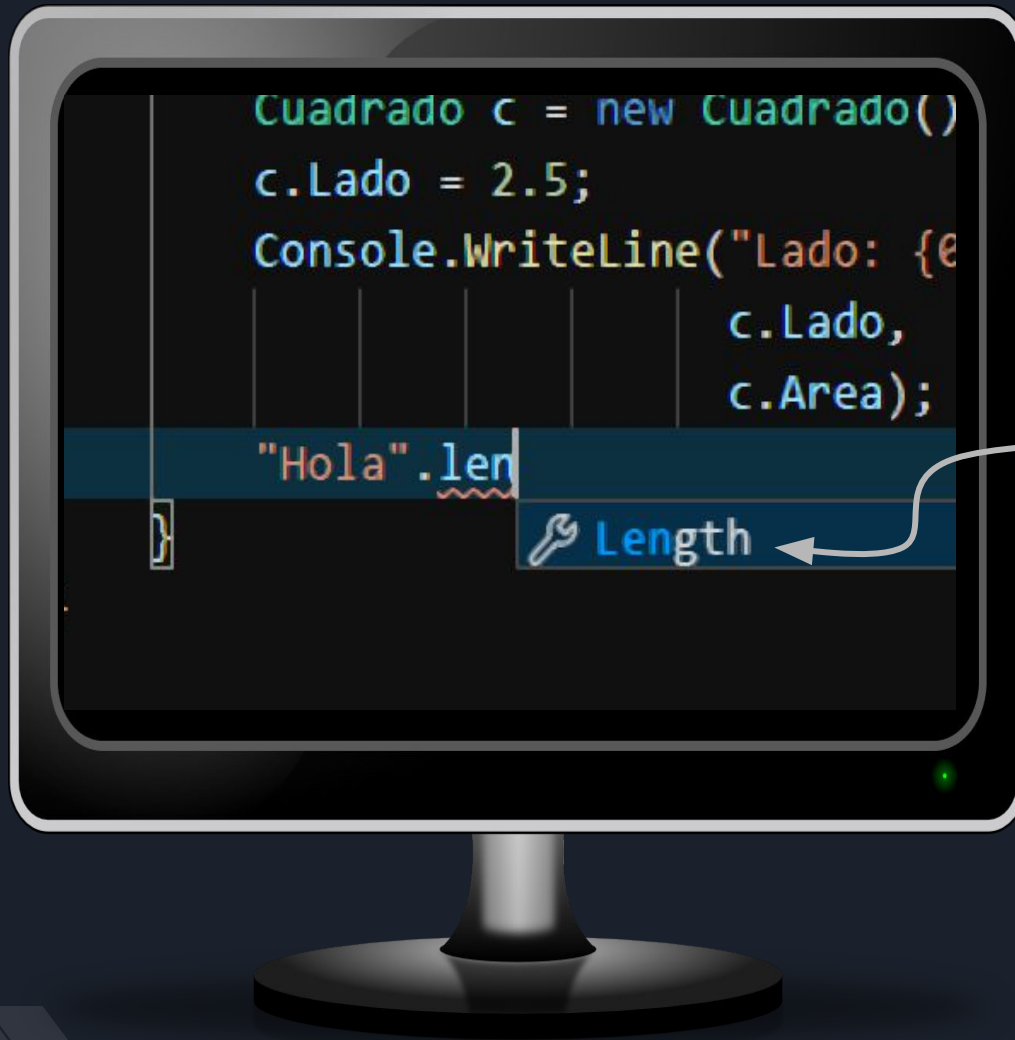
----- Cuadrado.cs -----

```
class Cuadrado {  
    private double _lado;  
    public double Lado {  
        get {  
            return _lado;  
        }  
        set {  
            _lado = value;  
        }  
    }  
    public double Area {  
        get {  
            return _lado * _lado;  
        }  
    }  
}
```



Lado: 2,5 área: 6,25





El autocompletado de código nos muestra muchas veces el ícono de las propiedades.

Las clases de la **BCL** no exponen campo públicos sino propiedades públicas, por ejemplo **Length** de un string, **Count** de una colección, no son campos sino propiedades

Descriptores de acceso con cuerpos de expresión

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;
}
```

Descriptores de acceso con cuerpos de expresión

En el caso de las **propiedades de sólo lectura**, las siguientes tres formas son equivalentes

```
public double Area
{
    get
    {
        return _lado * _lado;
    }
}
```

```
public double Area
{
    get => _lado * _lado;
}
```

```
public double Area => _lado * _lado;
```



Propiedades vs. Campos públicos

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.



Propiedades vs. Campos públicos

Por ejemplo:

```
public double Lado
{
    get => _lado;
    set => _lado = (value > 100) ? 100
                  : (value < 0) ? 0
                  : value;
}
```

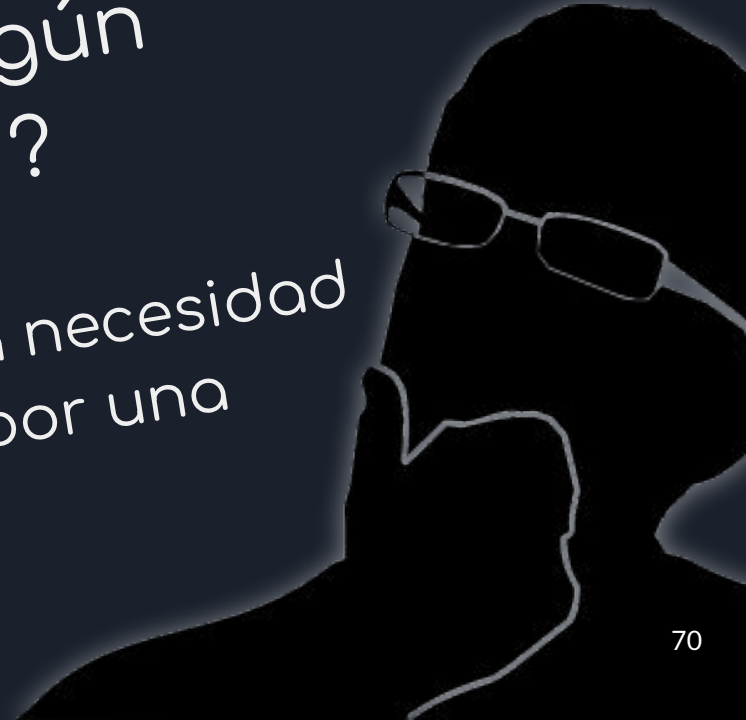
Esta propiedad
permite valores en
el intervalo [0,100]



Propiedades vs. Campos públicos

¿ Por qué deberíamos usar una propiedad pública en lugar de un campo público si no se necesita ningún procesamiento ?

En tal caso, cuando surja la necesidad podríamos cambiarlo por una propiedad



Propiedades vs. Campos públicos

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.



Propiedades implementadas automáticamente

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

```
class Persona
{
    private string _nombre;
    public string Nombre
    {
        get => _nombre;
        set => _nombre = value;
    }
}
```



backing field

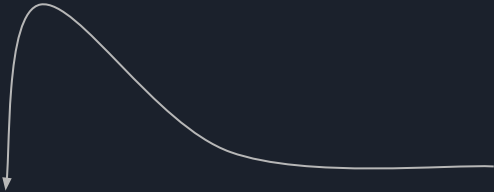
Propiedades implementadas automáticamente

- Para facilitar la tarea del programador C# 3.0 introdujo 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

Propiedades implementadas automáticamente

La clase **Persona** puede re-escribirse de la siguiente manera:

```
class Persona
{
    public string Nombre
    {
        get;
        set;
    }
}
```



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;
}
```

Clase con tres
campos públicos

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

Clase con tres
propiedades públicas
auto-implementadas



Propiedades implementadas automáticamente

Pueden inicializarse en su declaración:

```
public int Edad { get; set; } = 42;
```

También se pueden crear *propiedades readonly auto-implementadas*:

```
public string Version { get; } = "1.0.1";
```

Las *propiedades readonly auto-implementadas*, al igual que las variables *readonly*, pueden asignarse en su declaración o en cualquier constructor:

Propiedades implementadas automáticamente

Ejemplo de *propiedades readonly* auto-implementadas

Propiedades readonly
auto-implementada

```
class Persona
```

```
{
```

```
    . . .
```

```
    public string Id { get; } = Guid.NewGuid().ToString();
```

```
    public string Nombre { get; }
```

```
    public Persona(string nombre) => Nombre = nombre;
```

```
}
```

asignada en la
declaración

asignada en un
constructor

Propiedades estáticas

Las propiedades también pueden ser estáticas.
Ejemplo:

```
class Cuenta
```

```
{
```

```
    private int _monto;
```

```
    private static int s_total;
```

```
    public static int Total
```

```
{
```

```
        get => s_total;
```

```
        set => s_total = value;
```

```
}
```

```
    ...
```

```
}
```

Propiedad estática



Indizadores



Para presentar el concepto vamos a codificar dos clases, la clase Persona y la clase Familia de la siguiente manera:

```
namespace Teoria5;

class Persona
{
    public int Edad { get; }
    public string Nombre { get; }
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
    public void Imprimir() =>
        Console.WriteLine($"{Nombre} ({Edad})");
}
```





Para presentar el concepto vamos a codificar dos clases, la clase Persona y la clase Familia de la siguiente manera:

```
namespace Teoria5;
class Familia
{
    public Persona? Padre { get; set; }
    public Persona? Madre {get; set; }
    public Persona? Hijo { get; set; }
}
```





Codificar Program.cs y ejecutar

```
using Teoria5;  
Familia f = new Familia();  
f.Padre = new Persona("Juan", 50);  
f.Madre = new Persona("María", 40);  
f.Hijo = new Persona("José", 15);  
f.Madre.Imprimir();
```



Encapsulamiento - Indizadores

----- Program.cs -----

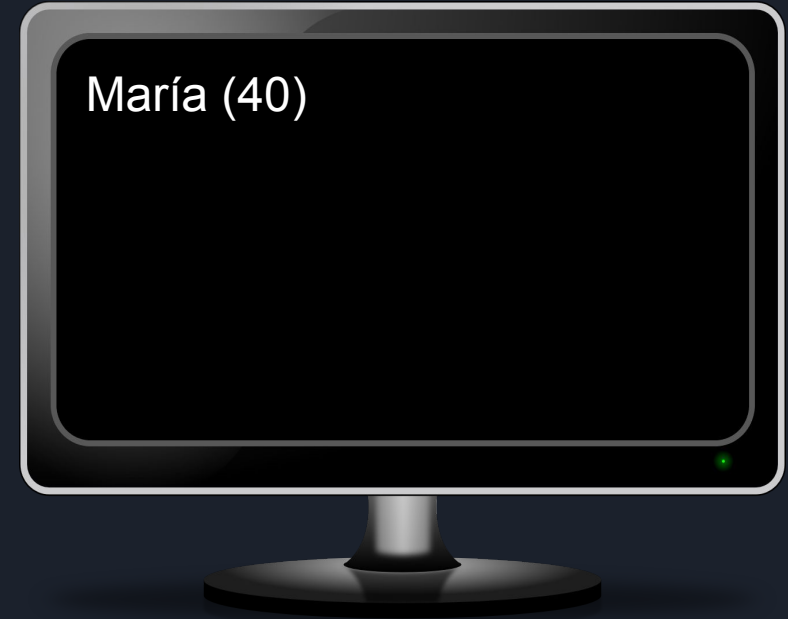
```
Familia f = new Familia();  
f.Padre = new Persona("Juan", 50);  
f.Madre = new Persona("María", 40);  
f.Hijo = new Persona("José", 15);  
f.Madre.Imprimir();
```

----- Persona.cs -----

```
class Persona {  
    public int Edad { get; }  
    public string Nombre { get; }  
    public Persona(string nombre, int edad)  
    {  
        Nombre = nombre;  
        Edad = edad;  
    }  
    public void Imprimir() =>  
        Console.WriteLine($"{Nombre} ({Edad})");  
}
```

----- Familia.cs -----

```
class Familia {  
    public Persona? Padre { get; set; }  
    public Persona? Madre { get; set; }  
    public Persona? Hijo { get; set; }  
}
```



Indizadores

Ahora se desea acceder a los miembros de una familia a través de un índice (como si se tratase de una colección). De esta forma queremos que:

`f.Padre` se acceda por medio de `f[0]`

`f.Madre` se acceda por medio de `f[1]`

`f.Hijo` se acceda por medio de `f[2]`

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 diferente (**sobrecarga**).

Los **indizadores** son **sólo de instancia**, no pueden definirse indizadores estáticos

Indizadores - Sintaxis

Tipo del
indizador

```
public Persona this[<lista de índices>]
{
    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 implícito
llamado **value** del
mismo tipo que el
indizador



Agregar el siguiente indizador de sólo lectura en la clase Familia

```
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;  
        }  
    }  
}
```





Modificar Program.cs y ejecutar

```
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();
}
```



Encapsulamiento - Indizadores

----- Program.cs -----

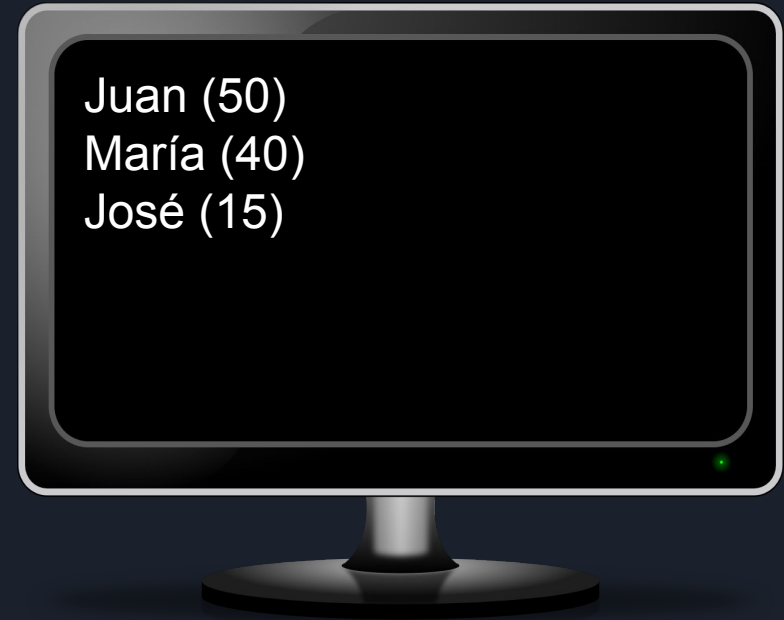
```
Familia f = new Familia();  
f.Padre = new Persona("Juan", 50);  
f.Madre = new Persona("María", 40);  
f.Hijo = new Persona("José", 15);  
for (int i = 0; i < 3; i++)  
{  
    f[i]?.Imprimir();  
}
```

----- Familia.cs -----

```
class Familia {  
    public Persona? Padre { get; set; }  
    public Persona? Madre { get; set; }  
    public Persona? Hijo { get; set; }  
    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;  
        }  
    }  
}
```

----- Persona.cs -----

```
class Persona  
{  
    . . .  
}
```





Codificar el bloque set del indizador para que sea de lectura/escritura

```
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;
    }
    set
    {
        if (i == 0) Padre = value;
        else if (i == 1) Madre = value;
        else if (i == 2) Hijo = value;
    }
}
```





Modificar el método Main para verificar el funcionamiento

```
using Teoria5;
Familia f = new Familia();
f.Padre = new Persona("Juan", 50);
f[1] = new Persona("María", 40);
f[2] = new Persona("José", 15);
for (int i = 0; i < 3; i++)
{
    f[i]?.Imprimir();
}
```



Encapsulamiento - Indizadores

----- Program.cs -----

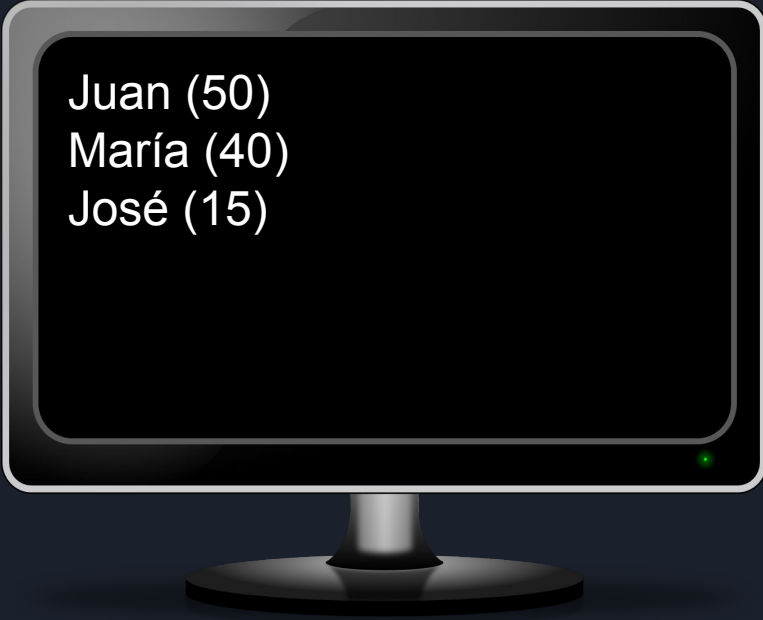
```
Familia f = new Familia();  
f.Padre = new Persona("Juan", 50);  
f[1] = new Persona("María", 40);  
f[2] = new Persona("José", 15);  
for (int i = 0; i < 3; i++)  
{  
    f[i]?.Imprimir();  
}
```

----- Familia.cs -----

```
class Familia {  
    public Persona? Padre { get; set; }  
    public Persona? Madre { get; set; }  
    public Persona? Hijo { get; set; }  
    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; }  
        set {  
            if (i == 0) Padre = value;  
            else if (i == 1) Madre = value;  
            else if (i == 2) Hijo = value; }  
    }  
}
```

----- Persona.cs -----

```
class Persona  
{  
    . . .  
}
```

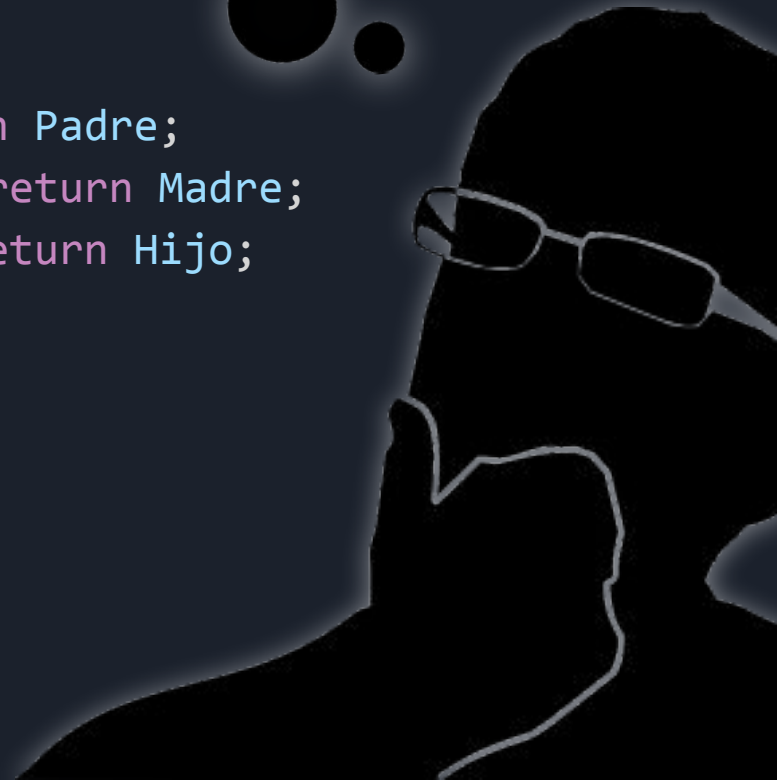


Juan (50)
María (40)
José (15)

Indizadores

¿Qué hace este
indizador?

```
public Persona? this[string st]
{
    get
    {
        if (Padre?.Nombre == st) return Padre;
        else if (Madre?.Nombre == st) return Madre;
        else if (Hijo?.Nombre == st) return Hijo;
        else return null;
    }
}
```



----- Program.cs -----

```
Familia f = new Familia();  
f.Padre = new Persona("Juan", 50);  
f[1] = new Persona("María", 40);  
f[2] = new Persona("José", 15);  
f["María"]?.Imprimir();
```

----- Familia.cs -----

```
class Familia {  
    public Persona? Padre { get; set; }  
    public Persona? Madre { get; set; }  
    public Persona? Hijo { get; set; }  
    public Persona? this[string st]  
    {  
        get  
        {  
            if (Padre?.Nombre == st) return Padre;  
            else if (Madre?.Nombre == st) return Madre;  
            else if (Hijo?.Nombre == st) return Hijo;  
            else return null;  
        }  
    }  
    . . .  
}
```

Devuelve el integrante
de la familia cuyo
nombre coincide con el
string usado como
índice



María (40)

Indizadores

¿Qué hace este
indizador?

```
public List<Persona> this[char c]
{
    get
    {
        List<Persona> result = new List<Persona>();
        if (Padre?.Nombre[0] == c) result.Add(Padre);
        if (Madre?.Nombre[0] == c) result.Add(Madre);
        if (Hijo?.Nombre[0] == c) result.Add(Hijo);
        return result;
    }
}
```

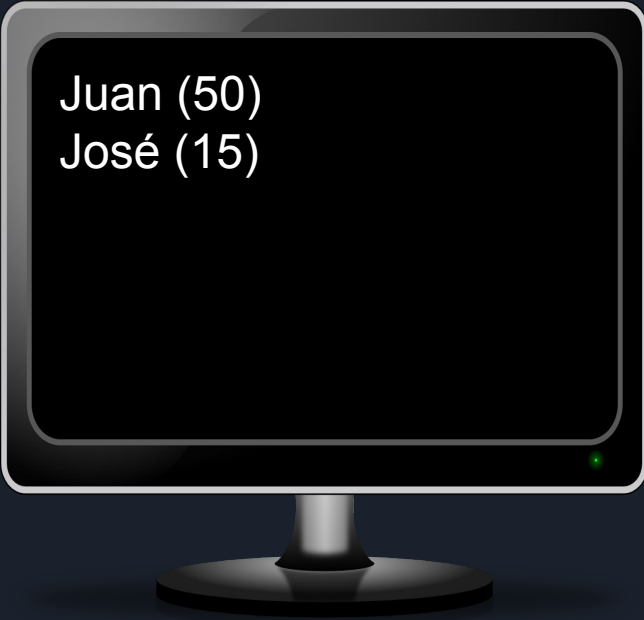
----- Program.cs -----

```
Familia f = new Familia();  
f.Padre = new Persona("Juan", 50);  
f[1] = new Persona("María", 40);  
f[2] = new Persona("José", 15);  
List<Persona> lista = f['J'];  
foreach(Persona p in lista) p.Imprimir();
```

----- Familia.cs -----

```
class Familia {  
    public Persona? Padre { get; set; }  
    public Persona? Madre { get; set; }  
    public Persona? Hijo { get; set; }  
    public List<Persona> this[char c]  
    {  
        get  
        {  
            List<Persona> result = new List<Persona>();  
            if (Padre?.Nombre[0] == c) result.Add(Padre);  
            if (Madre?.Nombre[0] == c) result.Add(Madre);  
            if (Hijo?.Nombre[0] == c) result.Add(Hijo);  
            return result;  
        }  
    }  
    ...  
}
```

Devuelve un ArrayList
con los integrantes de
la familia cuyos
nombres empiezan con
el carácter usado
como índice



Juan (50)
José (15)

Indizadores con múltiples índices. Ejemplo

```
class Tablero
{
    Dictionary<string, string> dic = new Dictionary<string, string>();
    public string this[char f, int c]
    {
        set
        {
            if (f >= 'A' && f <= 'J' && c >= 1 && c <= 10)
            {
                dic[f.ToString() + c] = value;
            }
        }
        get
        {
            try
            {
                return dic[f.ToString() + c];
            }
            catch
            {
                return "Agua";
            }
        }
    }
}
```

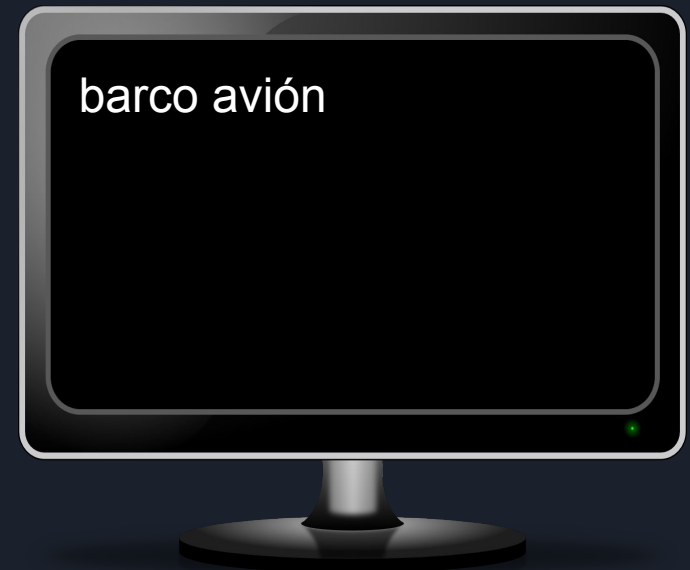
Encapsulamiento - Indizadores

----- Program.cs -----

```
Tablero t = new Tablero();  
t['H', 3] = "barco";  
t['A', 4] = "avión";  
Console.WriteLine($"{t['H', 3]} {t['A', 4]}");
```

----- Tablero.cs -----

```
class Tablero  
{  
    Dictionary<string, string> dic = new Dictionary<string, string>();  
    public string this[char f, int c]  
    {  
        set  
        {  
            if (f >= 'A' && f <= 'J' && c >= 1 && c <= 10)  
            {  
                dic[f.ToString() + c] = value;  
            }  
        }  
        get  
        {  
            try  
            {  
                return dic[f.ToString() + c];  
            }  
            catch  
            {  
                return "Agua";  
            }  
        }  
    }  
}
```



Algunas notas complementarias

Inicializadores de objeto

- Los **inicializadores de objeto** permiten asignar valores a cualquier campo o propiedad accesible de un objeto en el momento de su creación
- Un inicializador consiste en una lista de elementos separada por comas encerradas entre llaves **{ }** que sigue a la invocación del constructor
- Cada miembro de la lista mapea con un **campo o propiedad pública** del objeto al que le asigna un valor.

Inicializadores de objeto Ejemplo

```
public class Cat
{
    public int Edad { get; set; }
    public string? Nombre { get; set; }
    public Cat() { }
    public Cat(string nombre) => Nombre = nombre;
}
```

...

```
Cat c1 = new Cat() { Edad = 10, Nombre = "Fluffy" };
```

```
Cat c2 = new Cat("Fluffy") { Edad = 10 };
```

...



Inicializadores de objeto Ejemplo

También es posible hacer referencia a un indizador, por ejemplo utilizando la clase `Tablero` definida anteriormente

```
Tablero t = new Tablero()  
    { ['H', 3] = "barco", ['A', 4] = "avión" };
```

Encadenamiento de métodos (Fluent interface)

Dada la siguiente clase

```
class CuentaCorriente {  
    private double _monto = 0;  
    public void Depositar(double cantidad) {  
        _monto += cantidad;  
    }  
    ...  
}
```

Puede usarse de la siguiente forma:

```
...  
CuentaCorriente cc = new CuentaCorriente();  
cc.Depositar(100);  
cc.Depositar(200);  
cc.Depositar(30);  
...
```


Encadenamiento de métodos (Fluent interface)

El encadenamiento de métodos es una técnica muy útil que permite invocar múltiples métodos en una sentencia.

Para poder usarlo es necesario un pequeño cambio en la clase `CuentaCorriente`

```
class CuentaCorriente {  
    private double _monto = 0;  
    public CuentaCorriente Depositar(double cantidad)  
    {  
        _monto += cantidad;  
        return this;  
    }  
    ...  
}
```

En lugar de devolver
`void` ahora se devuelve
a sí mismo



Encadenamiento de métodos (Fluent interface)

```
class CuentaCorriente {  
    private double _monto = 0;  
    public CuentaCorriente Depositar(double cantidad)  
    {  
        _monto += cantidad;  
        return this;  
    }  
    ...  
}
```

Ahora es posible invocar al método de dos formas:
simple o en cadena:

```
...  
CuentaCorriente cc = new CuentaCorriente();  
cc.Depositar(100);  
cc.Depositar(200).Depositar(30).Depositar(15);  
...
```

Encadenamiento de métodos (Fluent interface)

La clase `StringBuilder` permite la invocación en cadena del método `Append(...)`

```
StringBuilder st = new StringBuilder();  
st.Append("Hola").Append(" ").Append("Mundo!");  
Console.WriteLine(st);
```



Hola Mundo!

Fin
de la teoría 5