



.NET

Teoría 4

Programación Orientada a Objetos



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.



Programación Orientada a objetos

- 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.

Classes

Clases

Qué es lo que tienen en común?



Modelo
Marca
Color
Velocidad

Acelerar
Desacelerar
Apagar
Arrancar

Atributos

Comportamiento

Una clase
encapsula atributos
y comportamientos
comunes

Codificando una clase en C#

Sintaxis:

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

Todos los métodos que definimos dentro de una clase son miembros de esa clase. Pero también hay otras categorías de miembros que iremos viendo en este curso



Codificando una clase en C#

1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria4`
4. Abrir `Visual Studio Code` sobre este proyecto





Crear el archivo fuente Auto.cs y codificar la clase Auto de la siguiente manera



File Edit Selection View Go Run Terminal Help



EXPLORER

> TEORIA4

> OUTLINE

> TIMELINE

✓ SOLUTION EXPLORER

✓ Teoria4

✓ Teoria4

> Depend...

Program.cs

Program.cs X

Program

1

2

3

Esta pestaña presenta una vista de la solución Teoria4 compuesta por un único proyecto denominado también Teoria4

Raíz de la Solución Teoria4

Raíz del Proyecto Teoria4



0 0 0



Projects: 1

Ln 3, Col 1

Spaces: 4

UTF-8 with BOM

CRLF

C#

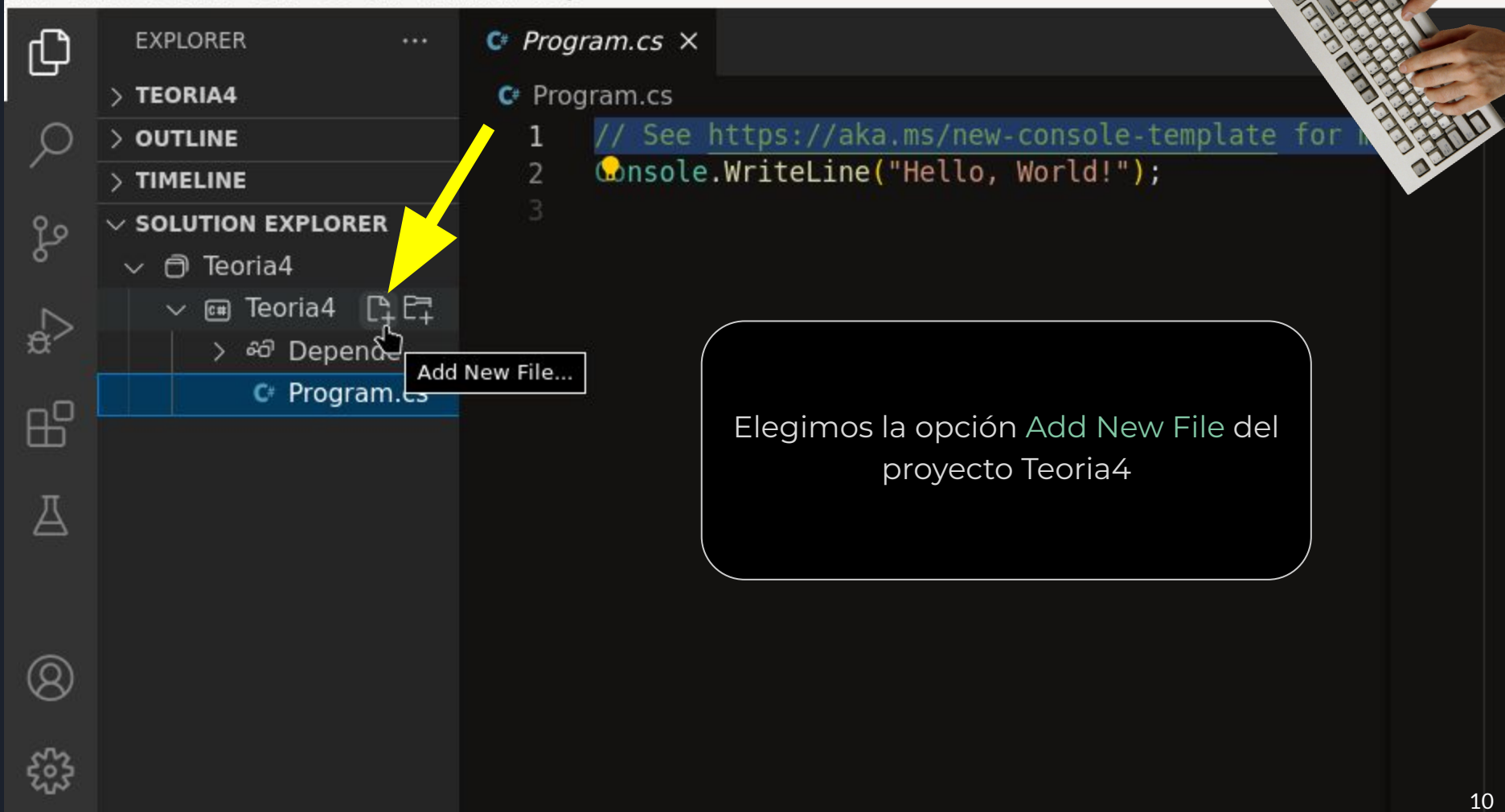




Crear el archivo fuente Auto.cs y codificar la clase Auto de la siguiente manera



File Edit Selection View Go Run Terminal Help



Elegimos la opción **Add New File** del proyecto Teoria4



Crear el archivo fuente Auto.cs y codificar la clase Auto de la siguiente manera



File Edit Selection View Go Run Terminal Help



EXPLORER

Select a template to create a new file

> TEORIA4

Class

> OUTLINE

Custom file (without template)

> TIMELINE

Enum

✓ SOLUTION EX

Interface

✓ Teoria4

MVC ViewImports

✓ Teo

MVC ViewStart

>

Protocol Buffer file

Razor component

Razor page

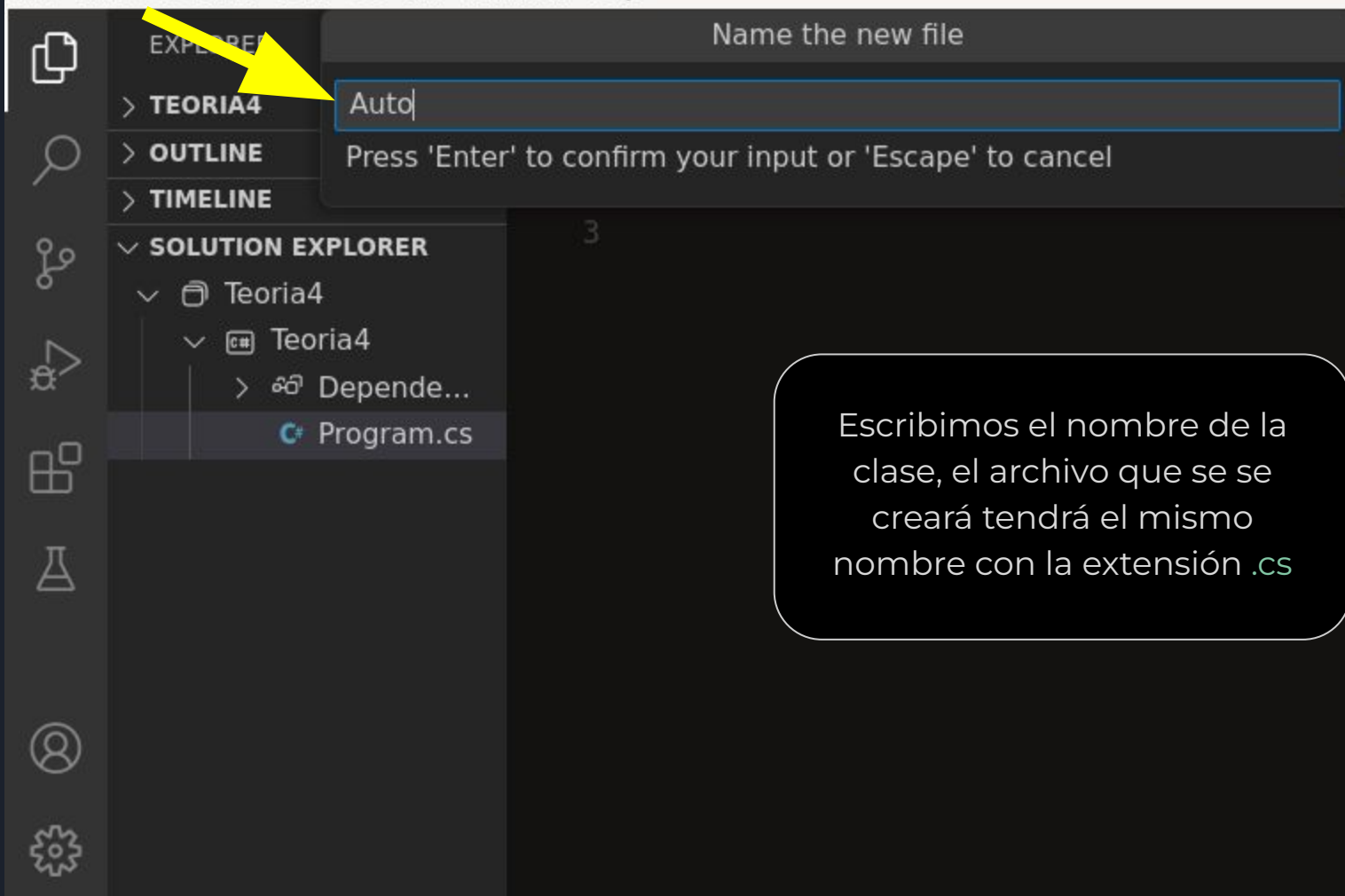
Elegimos como template la
opción **Class**



Crear el archivo fuente Auto.cs y codificar la clase Auto de la siguiente manera



File Edit Selection View Go Run Terminal Help



Escribimos el nombre de la clase, el archivo que se se creará tendrá el mismo nombre con la extensión .cs

clase Auto

File Edit Selection View Go Run Terminal Help

EXPLORER

- TEORIA4
- OUTLINE
- TIMELINE
- SOLUTION EXPLORER
 - Teoria4
 - Teoria4
 - Dependen...
 - C# Auto.cs**
 - C# Program.cs

C# Auto.cs

```
1 namespace Teoria4;
2
3 0 references
4 public class Auto
5 {
6 }
7
```

Es una buena práctica declarar nuestras clases dentro de un namespace que coincida con el nombre del proyecto

Clase `Auto` definida en el namespace `Teoria4`

Ln 7, Col 1 Spaces: 4 UTF-8 with BOM CRLF C#



Codificar Program.cs de la siguiente manera y ejecutar



```
using Teoria4;
```

```
Auto a;
```

Se declara una variable de tipo **Auto** pero aún no se ha instanciado ningún objeto

```
a = new Auto();
```

Se crea un objeto **Auto** (instanciación)

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

Se imprime el objeto en la consola

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto();
```

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

`Console.WriteLine(a)` imprime el tipo del objeto instanciado en `a` (incluido el namespace)

Este comportamiento se puede cambiar redefiniendo el método `ToString()` de la clase `Auto` (se verá más adelante en este curso)



Teoria4.Auto



Miembros de una Clase

Los miembros de una clase pueden ser:

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



Miembros de instancia

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

* Nota: las constantes se definen como miembros de instancia pero se utilizan como miembros estáticos (se verán en teoría 5)

Campos o variables de instancia



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** (posiblemente distinto al valor que tengan en dicho campo otros objetos de la misma clase)

Campos de instancia

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



Agregar los campos de instancia Marca y Modelo a la clase Auto

```
class Auto
{
    string? Marca;
    int Modelo;
}
```





Modificar Program.cs y ejecutar

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto();
```

```
a.Marca = "Nissan";
```

```
a.Modelo = 2017;
```

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





Modificar Program.cs

```
using Teoria4;
```

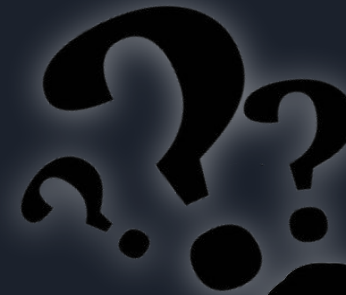
```
Auto a;
```

```
a = new Auto();
```

```
a.Marca = "Nissan";
```

```
a.Modelo = 2017;
```

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



¿Cuál es el
problema ?

Los miembros de una clase son privados de la clase por defecto.

```
class Auto
```

```
{
```

```
    string? Marca;
```

```
    int Modelo;
```

```
}
```

=

```
class Auto
```

```
{
```

```
    private string? Marca;
```

```
    private int Modelo;
```

```
}
```

Los campos **Marca** y **Modelo** son privados, por lo tanto sólo pueden accederse desde el código de la clase **Auto**, pero no es posible hacerlo desde fuera de esta clase





Agregar el modificador public en ambos campos

```
class Auto
{
    public string? Marca;
    public int Modelo;
}
```



No es bueno declarar
campos públicos.
Luego lo
arreglaremos



Modificar el método Main de la clase Program y ejecutar



```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto();
```

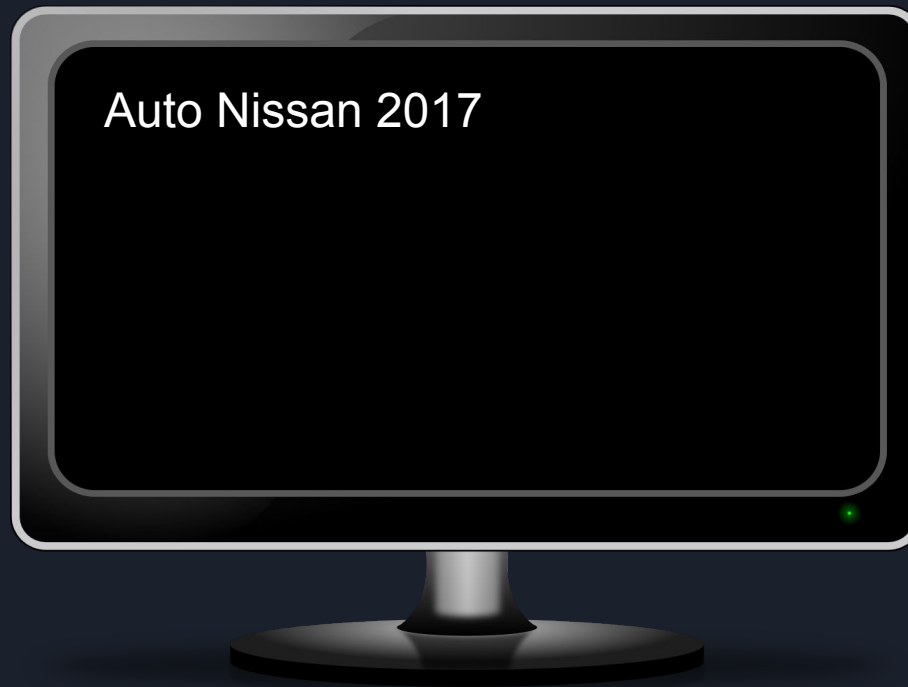
```
a.Marca = "Nissan";
```

```
a.Modelo = 2017;
```

```
Console.WriteLine($"Auto {a.Marca} {a.Modelo}");
```

Campos de instancia

```
using Teoria4;  
  
Auto a;  
a = new Auto();  
a.Marca = "Nissan";  
a.Modelo = 2017;  
Console.WriteLine($"Auto {a.Marca} {a.Modelo}");
```





Agregar las siguientes líneas:

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto();
```

```
a.Marca = "Nissan";
```

```
a.Modelo = 2017;
```

```
Console.WriteLine($"Auto {a.Marca} {a.Modelo}");
```

```
Auto b = new Auto();
```

```
b.Modelo = 2015;
```

```
b.Marca = "Ford";
```

```
Console.WriteLine($"Auto {b.Marca} {b.Modelo}");
```



```
using Teoria4;

Auto a;
a = new Auto();
a.Marca = "Nissan";
a.Modelo = 2017;
Console.WriteLine($"Auto {a.Marca} {a.Modelo}");
Auto b = new Auto();
b.Modelo = 2015;
b.Marca = "Ford";
Console.WriteLine($"Auto {b.Marca} {b.Modelo}");
```



Métodos de instancia



Métodos de instancia

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



Implementar el método `ObtenerDescripcion()` en la clase `Auto` para que los objetos autos devuelvan una descripción de sí mismos



```
class Auto
{
    public string? Marca;
    public int Modelo;
    public string ObtenerDescripcion()
    {
        return $"Auto {Marca} {Modelo}";
    }
}
```




Implementar el método `ObtenerDescripcion()` en la clase `Auto` para que los objetos `autos` devuelvan una descripción de sí mismos

```
class Auto
{
    public string? Marca;
    public int Modelo;
    public string ObtenerDescripcion() =>
        $"Auto {Marca} {Modelo}";
}
```

Así también funciona. De hecho hay cierta tendencia a usar `métodos con forma de expresión` cada vez que se pueda



Modificar Program.cs y ejecutar



```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto();
```

```
a.Marca = "Nissan";
```

```
a.Modelo = 2017;
```

```
Console.WriteLine(a.ObtenerDescripcion());
```

```
Auto b = new Auto();
```

```
b.Modelo = 2015;
```

```
b.Marca = "Ford";
```

```
Console.WriteLine(b.ObtenerDescripcion());
```

```
using Teoria4;

Auto a;
a = new Auto();
a.Marca = "Nissan";
a.Modelo = 2017;
Console.WriteLine(a.ObtenerDescripcion());
Auto b = new Auto();
b.Modelo = 2015;
b.Marca = "Ford";
Console.WriteLine(b.ObtenerDescripcion());
```

Le dimos a los objetos auto la responsabilidad de generar una descripción de sí mismos. Así evitamos tener que acceder a su representación interna para imprimirlos



Auto Nissan 2017
Auto Ford 2015

Constructores de instancia

Constructores de instancia

- Un **constructor de instancia** es un métodos 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

Constructores de instancia

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

Constructores de instancia

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



Modificar la clase Auto. Hacer privados sus campos



```
class Auto
```

```
{
```

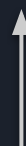
```
    → private string? _marca;
```

```
    → private int _modelo;
```

```
    public string ObtenerDescripcion() =>
```

```
        $"Auto {_marca} {_modelo}";
```

```
}
```





Modificar la clase Auto. Hacer privados sus campos

```
class Auto
```

```
{
```

```
    → private string? _marca;
```

```
    → private int _modelo;
```

```
    public string ObtenerDescripcion() =>
```

```
        $"Auto {_marca} {_modelo}";
```

```
}
```



La comunidad de .Net Core adoptó la convención de utilizar guión bajo al comienzo de un identificador de campo privado



Modificar la clase Auto. Agregar constructor.

```
class Auto
{
    private string? _marca;
    private int _modelo;
    public Auto(string marca, int modelo)
    {
        _marca = marca;
        _modelo = modelo;
    }
    public string ObtenerDescripcion() =>
        $"Auto {_marca} {_modelo}";
}
```





Modificar Program.cs y ejecutar.

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto("Nissan",2017);
```

```
Console.WriteLine(a.ObtenerDescripcion());
```

```
Auto b = new Auto("Ford",2015);
```

```
Console.WriteLine(b.ObtenerDescripcion());
```



```
using Teoria4;  
  
Auto a;  
a = new Auto("Nissan",2017);  
Console.WriteLine(a.ObtenerDescripcion());  
Auto b = new Auto("Ford",2015);  
Console.WriteLine(b.ObtenerDescripcion());
```



Auto Nissan 2017
Auto Ford 2015

Vamos mejorando nuestro código!

Estamos trabajando con objetos de la clase Auto pero su representación interna (campos) es inaccesible fuera de la clase.
A esto se lo conoce con el nombre de **Encapsulamiento**



Agregar las línea resaltada y ejecutar

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto("Nissan",2017);
```

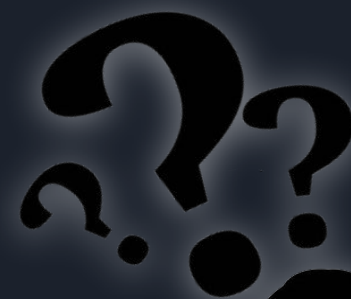
```
Console.WriteLine(a.ObtenerDescripcion());
```

```
Auto b = new Auto("Ford",2015);
```

```
Console.WriteLine(b.ObtenerDescripcion());
```

```
a = new Auto();
```

```
Console.WriteLine(a.ObtenerDescripcion());
```



¿Cuál es el problema ?

Constructores de instancia

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto("Nissan",2017);
```

```
Console.WriteLine(a.ObtenerDescripcion());
```

```
Auto b = new Auto("Ford",2015);
```

```
Console.WriteLine(b.ObtenerDescripcion());
```

```
a = new Auto(); Error de compilación
```

```
Console.WriteLine(a.ObtenerDescripcion());
```

Tenemos un error
porque la clase
Auto ya no
contiene un
constructor sin
parámetros

Constructores de instancia

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

```
public NombreClase()  
{  
}
```

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



*Si se define un
constructor
explícitamente,
el compilador
NO lo sobrecarga con el
constructor por defecto*

¿ Por qué ?



*Porque eventualmente
podemos querer forzar al
usuario a utilizar
determinado constructor
garantizando así la
consistencia de los objetos
creados.*





Quitar el signo “?” del tipo del campo `_marca`

```
class Auto
{
    private string? _marca;
    private int _modelo;

    . . .

}
```

cambiar `string?`
por `string`



Gracias al constructor, podemos asegurar que cada vez que se instancie un `Auto`, el campo `_marca` tendrá asignado un string válido



Constructores de instancia. Sobrecarga

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

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



Agregar el constructor sin parámetros para que al usarlo se instancie un Fiat modelo año actual

```
class Auto
{
    private string _marca;
    private int _modelo;
    public Auto(string marca, int modelo)
    {
        _marca = marca;
        _modelo = modelo;
    }
    public Auto()
    {
        _marca = "Fiat";
        _modelo = DateTime.Now.Year;
    }
}
```

. . .





Agregar el constructor sin parámetros para que al usarlo se instancie un Fiat modelo año actual

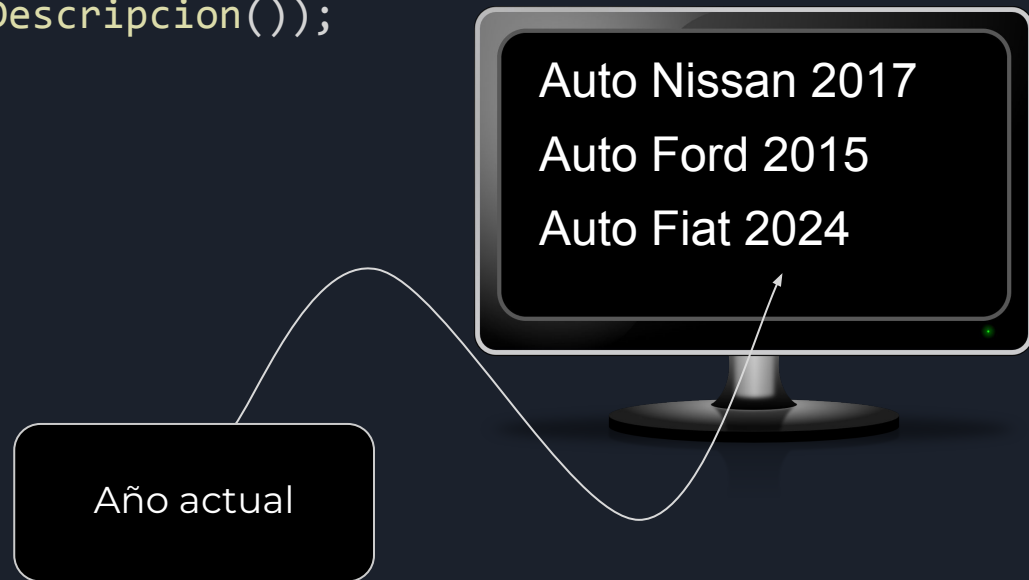
```
class Auto
{
    private string _marca;
    private int _modelo;
    public Auto(string marca, int
    {
        _marca = marca;
        _modelo = modelo;
    }
}
```

```
public Auto()
{
    _marca = "Fiat";
    _modelo = DateTime.Now.Year;
}
```

. . .

Acabamos de
sobrecargar al
constructor
definiendo dos
versiones distintas.
Ahora compila
sin errores.

```
using Teoria4;  
  
Auto a;  
a = new Auto("Nissan",2017);  
Console.WriteLine(a.ObtenerDescripcion());  
Auto b = new Auto("Ford",2015);  
Console.WriteLine(b.ObtenerDescripcion());  
a = new Auto();  
Console.WriteLine(a.ObtenerDescripcion());
```



Constructores de instancia. Sobrecarga

- En el encabezado de un constructor se puede invocar a otro constructor de la misma clase empleando la sintaxis `:this`
- Este constructor invocado se ejecuta antes que las instrucciones del cuerpo del constructor invocador.



Agregar un constructor a la clase Auto que reciba la marca como parámetro. El modelo del auto creado debe ser igual al año actual.

```
. . .  
public Auto()  
{  
    _marca = "Fiat";  
    _modelo = DateTime.Now.Year;  
}
```

```
public Auto(string marca) : this()  
{  
    _marca = marca;  
}
```

```
. . .
```





Agregar un constructor a la clase Auto que reciba la marca como parámetro. El modelo del auto creado debe ser igual al año actual.

```
. . .  
  
public Auto()  
{  
    _marca = "Fiat";  
    _modelo = DateTime.Now.Year;  
}  
  
public Auto(string marca) : this()  
{  
    _marca = marca;  
}  
  
. . .  
}
```

Invocación



Modificar el método Main para utilizar este constructor. Ejecutar para probar su funcionamiento

```
using Teoria4;
```

```
Auto a;
```

```
a = new Auto("Nissan",2017);
```

```
Console.WriteLine(a.ObtenerDescripcion());
```

```
Auto b = new Auto("Ford",2015);
```

```
Console.WriteLine(b.ObtenerDescripcion());
```

```
a = new Auto("Renault");
```

```
Console.WriteLine(a.ObtenerDescripcion());
```



```
Auto Nissan 2017  
Auto Ford 2015  
Auto Renault 2024
```

Constructores de instancia. Sobrecarga

El último constructor también se puede codificar de la siguiente forma:

```
public Auto(string marca):this(marca, DateTime.Now.Year)
{
}
```

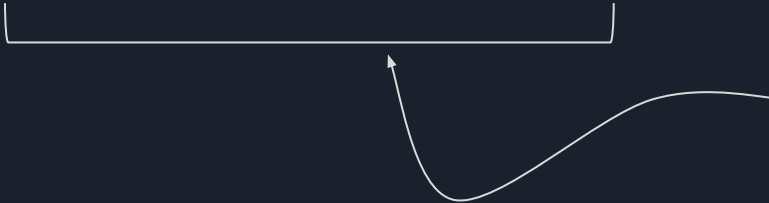
El cuerpo está vacío,
todo se resuelve en
la invocación al otro
constructor



Constructores principales (o primarios) incorporados a partir de C# 12

- A partir de C# 12 Se admiten parámetros en el encabezado de la clase

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



Estos parámetros están
accesibles para lectura y
escritura en todo el
cuerpo de la clase

- Los parámetros del constructor primario pueden utilizarse para la inicialización o directamente como estado del objeto.
- Todos los constructores declarados explícitamente deben llamar al constructor principal mediante la sintaxis `this(...)`

Constructores principales (o primarios) incorporados a partir de C# 12

- En el siguiente ejemplo estamos utilizando un parámetro para inicializar un campo privado y otro directamente como estado del objeto

```
public class Auto(string marca, int modelo)
{
    private string _marca = marca;

    public string GetMarca()
    {
        return _marca;
    }

    public int GetModelo()
    {
        return modelo;
    }
    ...
}
```

El parámetro `marca` se utiliza para inicializar el campo `_marca`

El parámetro `modelo` se utiliza directamente como estado del objeto

Utilizar constructores principales permite economizar líneas de código

```
class Auto
{
    private string _marca;
    private int _modelo;
    public Auto(string marca, int modelo)
    {
        _marca = marca;
        _modelo = modelo;
    }
    public Auto()
    {
        _marca = "Fiat";
        _modelo = DateTime.Now.Year;
    }
    public Auto(string marca) : this()
    {
        _marca = marca;
    }
    public string ObtenerDescripcion() =>
        $"Auto {_marca} {_modelo}";
}
```



```
public class Auto(string marca, int modelo)
{
    public Auto() : this("Fiat",
        DateTime.Now.Year) { }
    public Auto(string marca) : this(marca,
        DateTime.Now.Year) { }
    public string ObtenerDescripcion() =>
        $"Auto {marca} {modelo}";
}
```



Métodos de instancia. Sobrecarga

- Los métodos también pueden ser sobrecargados
- Para sobrecargar los métodos valen las mismas consideraciones que en el caso de los constructores
- El valor de retorno NO puede utilizarse como única diferencia para permitir una sobrecarga

Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas válidas

```
void procesar()
```

```
void procesar(int valor)
```

```
void procesar(float valor)
```

```
void procesar(double valor)
```

```
void procesar(int valor1, double valor2)
```

```
void procesar(double valor1, int valor2)
```

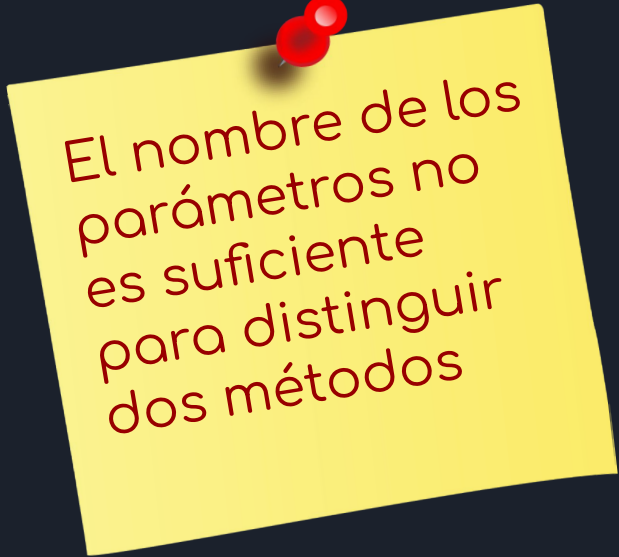
```
void procesar(out double valor)
```


Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

```
void procesar(int valor1)
```

```
void procesar(int valor2)
```



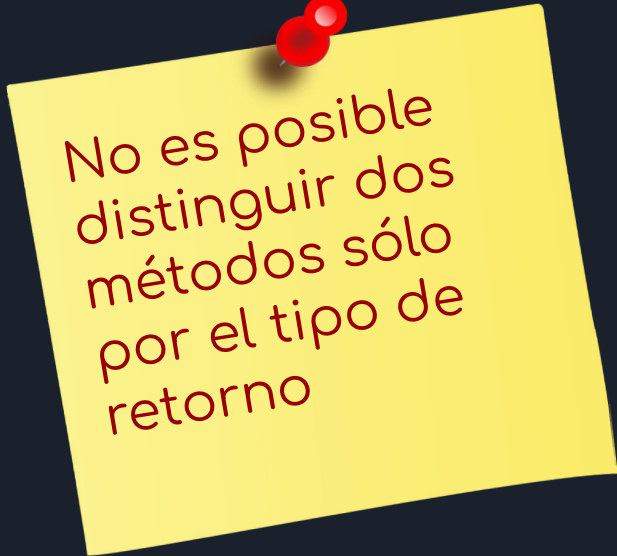
El nombre de los
parámetros no
es suficiente
para distinguir
dos métodos

Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

```
void procesar(int valor)
```

```
int procesar(int valor)
```



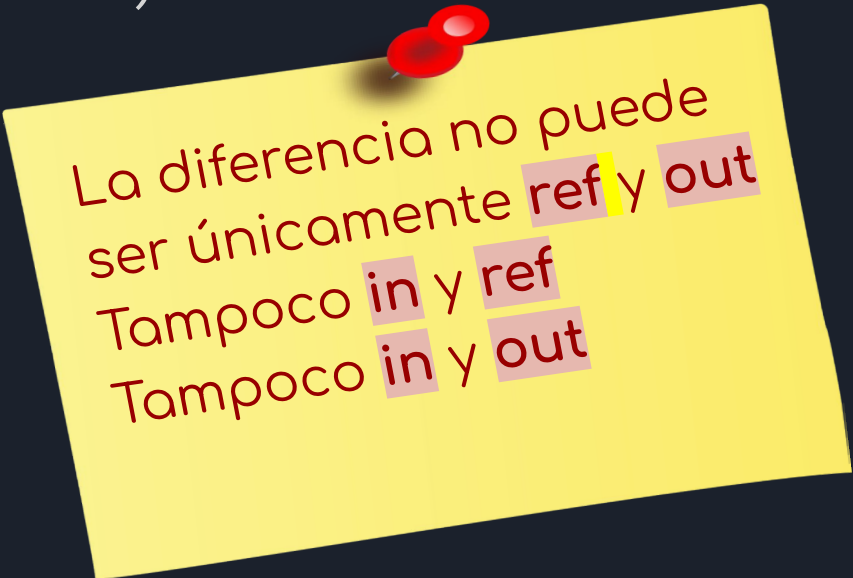
No es posible
distinguir dos
métodos sólo
por el tipo de
retorno

Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

```
void procesar(ref int valor1)
```

```
void procesar(out int valor2)
```

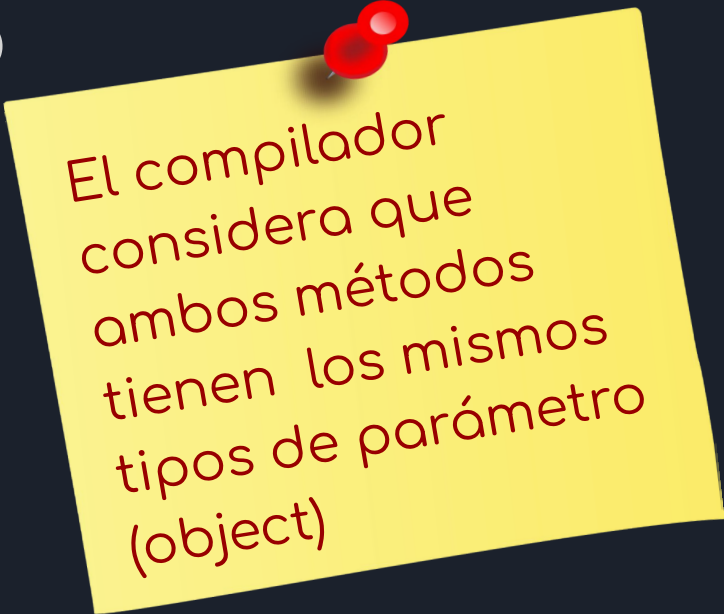


La diferencia no puede
ser únicamente **ref** y **out**
Tampoco **in** y **ref**
Tampoco **in** y **out**


Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

```
void procesar(object valor1)  
void procesar(dynamic valor2)
```



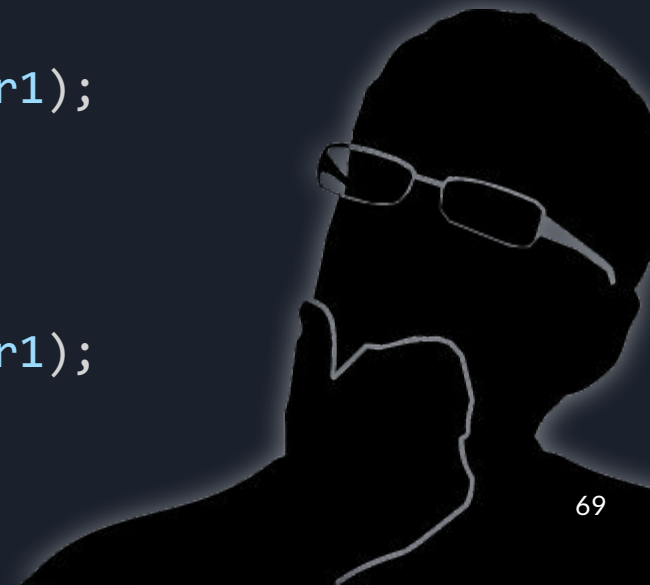
El compilador considera que ambos métodos tienen los mismos tipos de parámetro (object)



Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
s.Procesar(12);  
s.Procesar(12.1);
```

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(int valor1)  
    {  
        Console.WriteLine("entero " + valor1);  
    }  
}
```



Métodos de instancia - Sobrecarga

```
Sobrecarga s = new Sobrecarga();  
s.Procesar(12);  
s.Procesar(12.1);
```

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(int valor1)  
    {  
        Console.WriteLine("entero " + valor1);  
    }  
}
```

Si existe más de una versión del método que sea elegible se invoca la más específica

```
entero 12  
objeto 12,1
```

Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
object o = 12;  
s.Procesar(o);  
o = 12.1;  
s.Procesar(o);
```

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(int valor1)  
    {  
        Console.WriteLine("entero " + valor1);  
    }  
}
```

Métodos de instancia - Sobrecarga

```
Sobrecarga s = new Sobrecarga();  
object o = 12;  
s.Procesar(o);  
o = 12.1;  
s.Procesar(o);
```

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(int valor1)  
    {  
        Console.WriteLine("entero " + valor1);  
    }  
}
```

Si no está involucrado un tipo dynamic, la resolución de la sobrecarga se realiza en tiempo de compilación

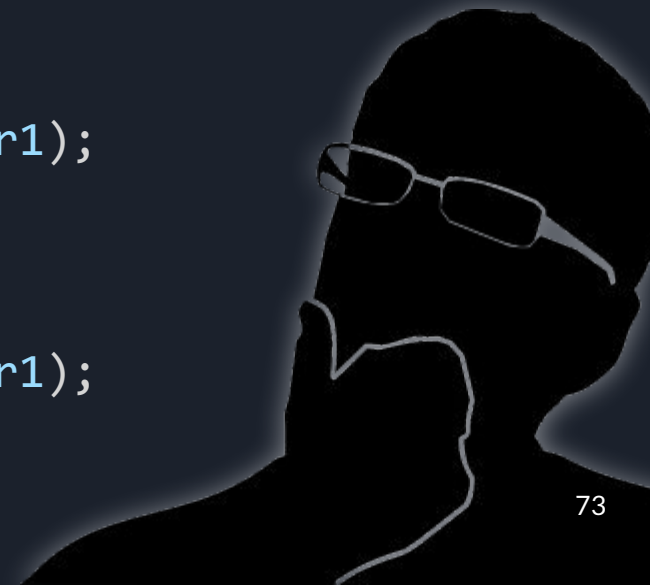


```
objeto 12  
objeto 12,1
```


Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
dynamic o = 12;  
s.Procesar(o);  
o = 12.1;  
s.Procesar(o);
```

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(int valor1)  
    {  
        Console.WriteLine("entero " + valor1);  
    }  
}
```



Métodos de instancia - Sobrecarga

```
Sobrecarga s = new Sobrecarga();  
dynamic o = 12;  
s.Procesar(o);  
o = 12.1;  
s.Procesar(o);
```

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(int valor1)  
    {  
        Console.WriteLine("entero " + valor1);  
    }  
}
```

Como el argumento que se envía a procesar es de tipo **dynamic**, la resolución de la sobrecarga se produce en tiempo de ejecución

```
entero 12  
objeto 12,1
```

Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
byte b = 12;  
s.Procesar(b);  
int i = 1000;  
s.Procesar(i);  
double d = 0.125;  
s.Procesar(d);
```

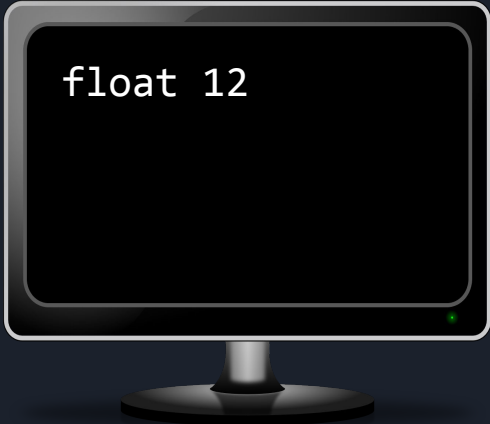
```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(float valor1)  
    {  
        Console.WriteLine("float " + valor1);  
    }  
}
```

Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
byte b = 12;  
s.Procesar(b);  
int i = 1000;  
s.Procesar(i);  
double d = 0.125;  
s.Procesar(d);
```

Porque existe
conversión
implícita de byte a
float

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(float valor1)  
    {  
        Console.WriteLine("float " + valor1);  
    }  
}
```



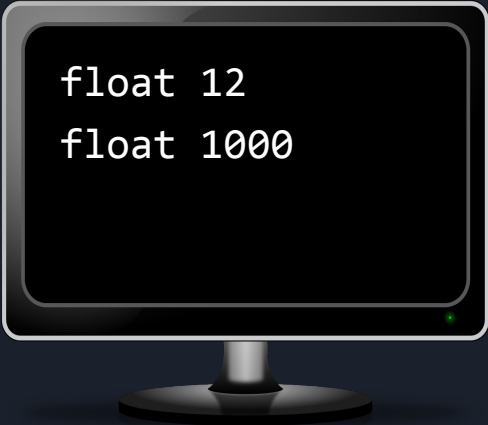
float 12

Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
byte b = 12;  
s.Procesar(b);  
int i = 1000;  
s.Procesar(i);  
double d = 0.125;  
s.Procesar(d);
```

Porque existe
conversión
implícita de int a
float

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(float valor1)  
    {  
        Console.WriteLine("float " + valor1);  
    }  
}
```




```
float 12  
float 1000
```

Para pensar
¿Cuál es la salida por consola?

```
Sobrecarga s = new Sobrecarga();  
byte b = 12;  
s.Procesar(b);  
int i = 1000;  
s.Procesar(i);  
double d = 0.125;  
s.Procesar(d);
```

No existe conversión
implícita de double a
float

```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(float valor1)  
    {  
        Console.WriteLine("float " + valor1);  
    }  
}
```



```
float 12  
float 1000  
objeto 0,125
```

Arquitectura modular

Arquitectura NO modular vs. arquitectura modular

- **Arquitectura NO modular:** Único módulo ejecutable que implementamos en un único proyecto (lo que hicimos hasta ahora)
- **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 a los **ejecutables**, como a las **bibliotecas de clases** reciben el nombre de **ensamblados** en **.Net**



Codificando una solución modular para un Estacionamiento

1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear una `solución` llamada `Estacionamiento` con el siguiente comando:

```
dotnet new sln -o Estacionamiento
```





Codificando una solución modular para un Estacionamiento

Moverse a la carpeta **Estacionamiento**:

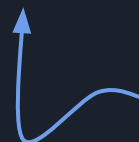
```
cd Estacionamiento
```

Crear la biblioteca de clases **Automotores**:

```
dotnet new classlib -o Automotores
```

Crear la aplicación de consola **ConsolaUI**:

```
dotnet new console -o ConsolaUI
```



UI por “User Interface”. Será la interfaz de usuario, un ejecutable, en este caso, una aplicación de consola





Codificando una solución modular para un Estacionamiento

Agregar ambos proyectos a la solución

```
dotnet sln add ./Automotores
```

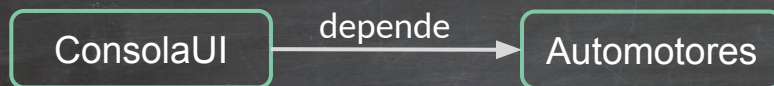
```
dotnet sln add ./ConsolaUI
```

Agregar en **ConsolaUI** una referencia a **Automotores**:

```
dotnet add ./ConsolaUI reference ./Automotores
```



Acabamos de crear una dependencia



Para su ejecución, **ConsolaUI** necesita la biblioteca **Automotores**. Pero **Automotores**, no necesita en absoluto a **ConsolaUI** (podemos compartirla con otras soluciones)



Codificando una solución modular para un Estacionamiento

Agregar ambos proyectos a la solución

```
dotnet sln add ./Automotores
```

```
dotnet sln add ./ConsolaUI
```

Agregar en **ConsolaUI** una referencia a **Automotores**:

```
dotnet add ./ConsolaUI reference ./Automotores
```

Abrir Visual Studio Code

```
code .
```



Arquitectura modular

File Edit Selection View Go Run Terminal Help



The screenshot shows the Visual Studio IDE with the following components:

- EXPLORER:** Displays a solution structure with folders 'ESTACIONAMIENTO', 'OUTLINE', and 'TIMELINE'. Under 'SOLUTION EXPLORER', there is a folder 'Estacionamiento' containing 'Automotores', and a folder 'ConsolaUI' containing 'Dependencies' and 'Program.cs'. The 'ConsolaUI' folder is selected.
- ConsolaUI.csproj:** The main editor shows the XML content of the project file. The first line is `Project Sdk="Microsoft.NET.Sdk">`. The second line is `<ItemGroup>`. The third line is `<ProjectReference Include="..\Automotores\Automotores.csproj" />`. The fourth line is `</ItemGroup>`. The fifth line is `<PropertyGroup>`. The sixth line is `<OutputType>Exe</OutputType>`. The seventh line is `<TargetFramework>net8.0</TargetFramework>`. The eighth line is `<ImplicitUsings>enable</ImplicitUsings>`. The ninth line is `<Nullable>enable</Nullable>`. The tenth line is `</PropertyGroup>`. The eleventh line is `Project>`. The twelfth line is `</Project>`. The thirteenth line is `</Project>`. The fourteenth line is `</Project>`. The fifteenth line is `</Project>`.

A callout box with a white border and rounded corners contains the text: "Observar que el proyecto de consola **ConsolaUI** tiene una referencia a la biblioteca de clases **Automotores**". A white arrow points from the text 'Automotores' in the callout box to the 'Automotores.csproj' file in the project reference line of the code.


Observar que el proyecto de consola **ConsolaUI** tiene una referencia a la biblioteca de clases **Automotores**

< 0 0 0 Projects: 2

Ln 1, Col 1 Spaces: 2 UTF-8 with BOM CRLF XML

Arquitectura modular

File Edit Selection View Go Run Terminal Help



The screenshot shows the Visual Studio IDE with a C# project named 'Estacionamiento'. The 'SOLUTION EXPLORER' on the left shows the project structure: 'Estacionamiento' (solution) contains 'Automotores' (project), which contains 'Class1.cs' (file) and 'ConsolaUI' (project). 'Class1.cs' is selected. The main editor shows the code for 'Class1.cs' with the following content:

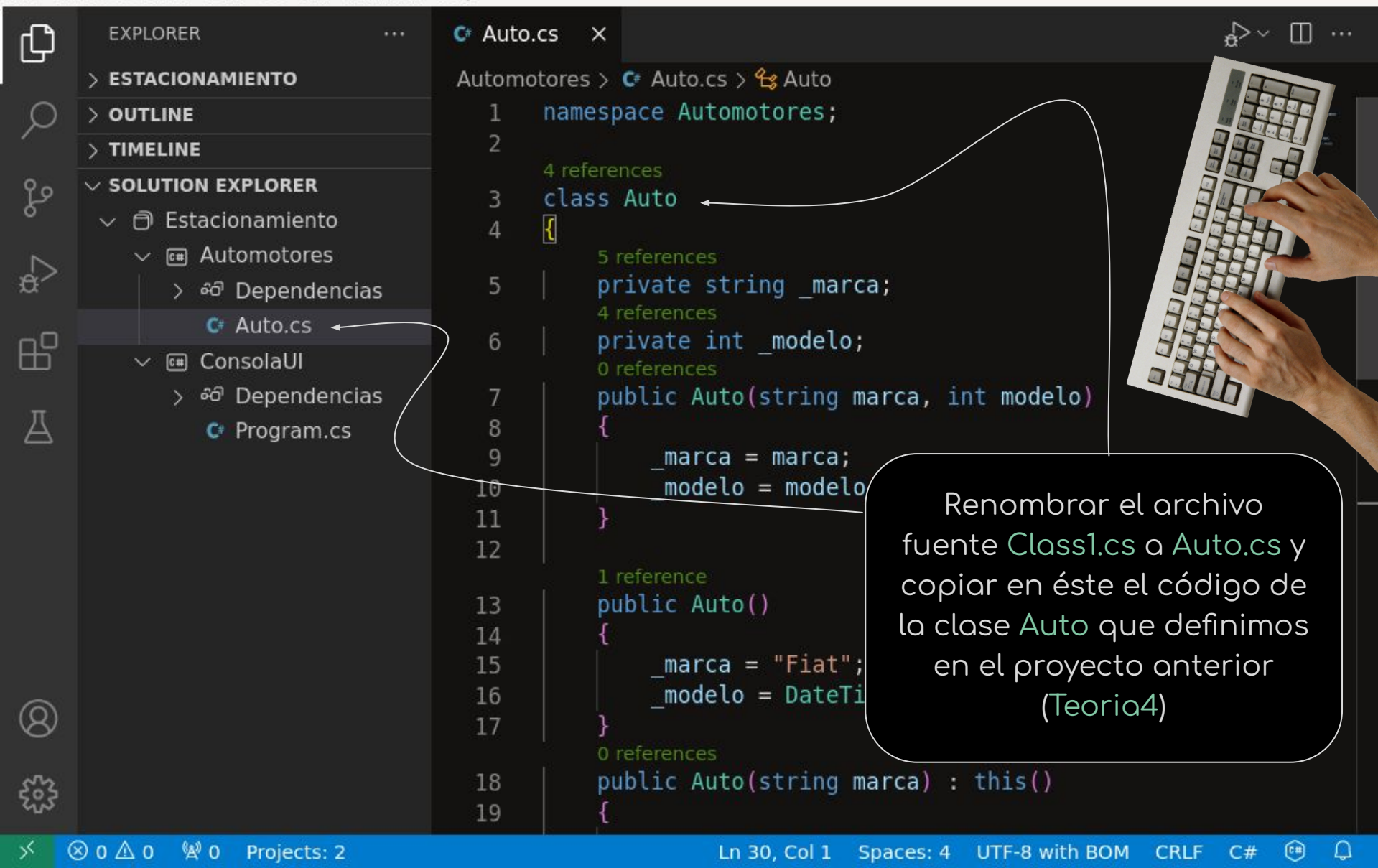
```
1 namespace Automotores;  
2  
3 public class Class1  
4 {  
5  
6 }  
7
```

A callout box points to the 'Class1' class name in the code, containing the text: 'Al crear una biblioteca de clases se crea por defecto la clase pública Class1 (habitualmente se borra o renombra adecuadamente)'.

At the bottom of the IDE, the status bar shows: 'Ln 1, Col 1 Spaces: 4 UTF-8 with BOM CRLF C#'.

Arquitectura modular

File Edit Selection View Go Run Terminal Help



EXPLORER

- > ESTACIONAMIENTO
- > OUTLINE
- > TIMELINE
- ✓ SOLUTION EXPLORER
 - ✓ Estacionamiento
 - ✓ Automotores
 - > Dependencias
 - Auto.cs
 - ✓ ConsolaUI
 - > Dependencias
 - Program.cs

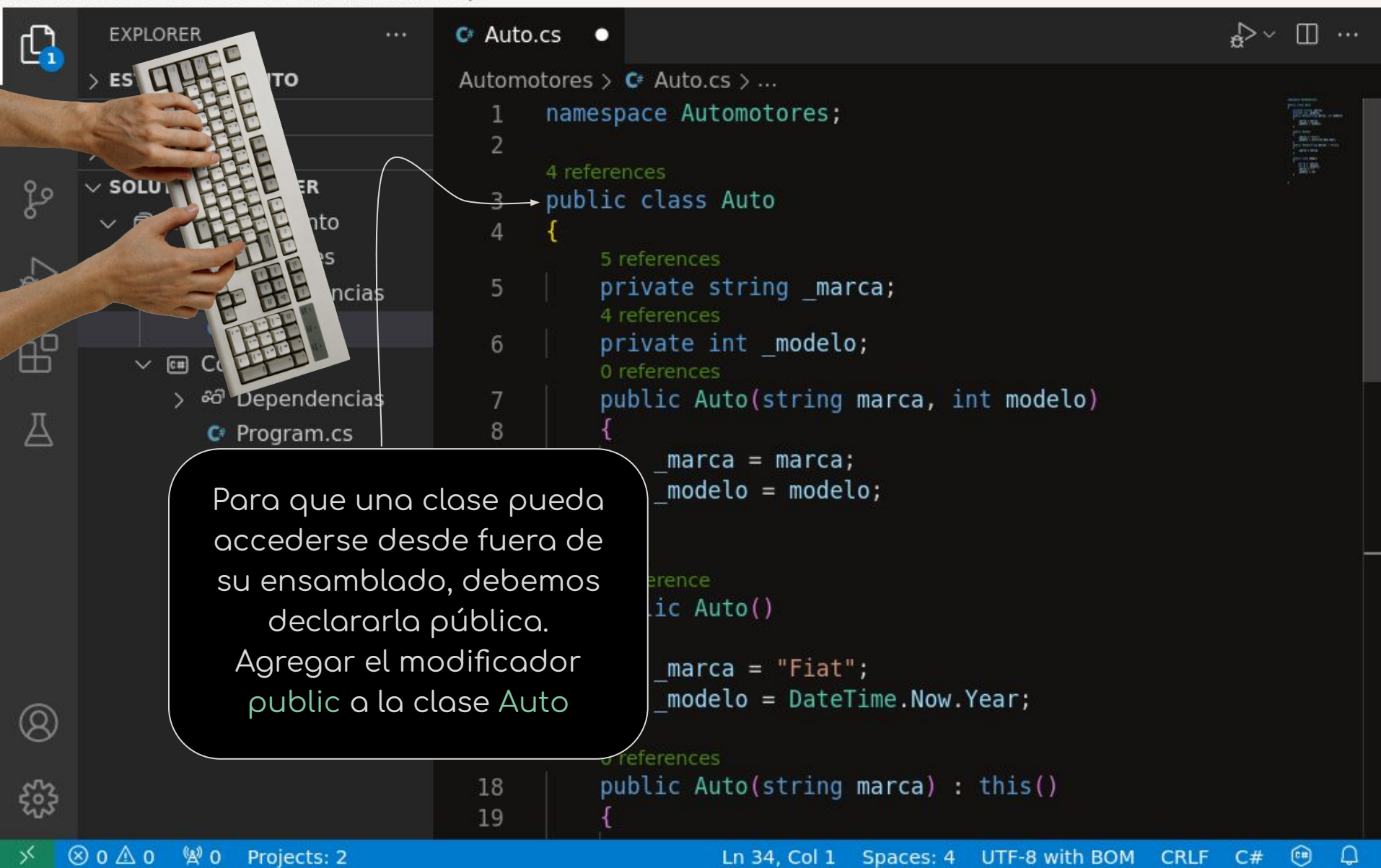
Auto.cs

```
1 namespace Automotores;
2
3 class Auto
4 {
5     private string _marca;
6     private int _modelo;
7     public Auto(string marca, int modelo)
8     {
9         _marca = marca;
10        modelo = modelo;
11    }
12
13    public Auto()
14    {
15        _marca = "Fiat";
16        _modelo = DateTime.Now.Year;
17    }
18    public Auto(string marca) : this()
19    {
```

Renombrar el archivo
fuente `Class1.cs` a `Auto.cs` y
copiar en éste el código de
la clase `Auto` que definimos
en el proyecto anterior
(Teoria4)

Arquitectura modular

File Edit Selection View Go Run Terminal Help



EXPLORER

ES TO

SOLU

nto

es

ncias

Co

Dependencias

Program.cs

Auto.cs

Automotores > Auto.cs > ...

```
1 namespace Automotores;
2
3 public class Auto
4 {
5     private string _marca;
6     private int _modelo;
7     public Auto(string marca, int modelo)
8     {
9         _marca = marca;
10        _modelo = modelo;
11    }
12    public static Auto()
13    {
14        _marca = "Fiat";
15        _modelo = DateTime.Now.Year;
16    }
17    public Auto(string marca) : this()
18    {
19    }
```

4 references

5 references

4 references

0 references

reference

0 references

Para que una clase pueda accederse desde fuera de su ensamblado, debemos declararla pública. Agregar el modificador **public** a la clase **Auto**

Ln 34, Col 1 Spaces: 4 UTF-8 with BOM CRLF C#



Codificar Program.cs del proyecto ConsolaUI para verificar que es posible acceder a las clases públicas de la biblioteca creada

```
using Automotores;
```

```
Auto a = new Auto("Renault",2020);
```

```
Console.WriteLine(a.ObtenerDescripcion());
```



Algunas notas complementarias



Nota sobre invocación a métodos y constructores

Los **métodos** (aunque devuelvan valores) y los **constructores** de objetos (expresiones con operador **new**) **pueden usarse como una instrucción**, es decir, no se requiere asignar el valor devuelto, en todo caso, dicho valor se pierde.

Ejemplo:

```
void Prueba()  
{  
    "hola".ToUpper();  
    new System.Text.StringBuilder("C#");  
    int i;  
    i;  
    "hola".Length;  
    int tamaño = "hola".Length;  
    Console.Write("hola".Length);  
}
```

Válido. El método **ToUpper()** devuelve "HOLA" pero este valor se pierde

Válido. Se está instanciando un objeto **StringBuilder**, pero una vez creado se pierde su referencia

ERROR DE COMPILACIÓN. las variables y las propiedades no pueden utilizarse como si fuesen instrucciones.
Length no es un método, es una propiedad de la clase **string**

Válido. El valor de la propiedad **Length** no se pierde, lo estamos utilizando

El límite del encapsulamiento es la clase (no la instancia)

```
class Persona {  
    private string? nombre;  
    public bool MeLlamoIgual(Persona p) {  
        return (this.nombre == p.nombre);  
    }  
}
```

OK Dentro del código de Persona se puede acceder al campo privado de cualquier objeto Persona

```
class Animal {  
    private string? nombre;  
    public bool MeLlamoIgual(Persona p) {  
        return (this.nombre == p.nombre);  
    }  
}
```

ERROR DE COMPILACIÓN No se puede acceder al campo privado de un objeto Persona fuera del código de la clase Persona

Miembros de instancia. Uso de `this`

Dentro de un `constructor` o `método de instancia`, la palabra clave `this` hace referencia a la instancia (el propio objeto) que está ejecutando ese código.


Puede ser útil para `diferenciar` el nombre de un `campo` de una `variable local` o `parámetro` con el mismo nombre



Miembros de instancia. Uso de `this`

Ejemplo

```
class Persona {  
    int edad;  
    string nombre;  
    public void Actualizar(string nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
        this.Notificar();  
    }  
    private void Notificar() {  
        ...  
    }  
}
```



Este `this` puede omitirse

?? Operador null-coalescing (operador de fusión nula)

`a = b ?? c;`

equivale a

`a = (b != null) ? b : c;`



```
string st1 = null;  
string st2 = "chau";  
string st = st1 ?? "hola";  
Console.WriteLine(st);  
st = st2 ?? "hola";  
Console.WriteLine(st);  
st = null ?? null ?? "ABC" ?? "123";  
Console.WriteLine(st);
```

Se pueden encadenar. Se devuelve el primer valor no nulo encontrado



hola
chau
ABC

??= Asignación null-coalescing

(disponible a partir de C# 8.0)

a ??= b;

equivale a

a = a ?? b;



Operador condicional NULL (?. y ?[])

```
string? nombre = persona?.Nombre;
```

Si la variable `persona` es `null`, en lugar de generar una excepción `NullReferenceException`, se cortocircuita y devuelve `null`. A menudo se utiliza con el operador `??`

```
string nombre = persona?.Nombre ?? "indefinido";
```

También se usa para invocar métodos o acceder a un elemento de una colección de forma condicional por ejemplo:

```
cuenta?.Depositar(1000);  
Console.WriteLine(vector?[1]);
```

Sólo se invoca `Depositar` si
`cuenta != null`
Sólo se accede al elemento del
vector si `vector != null`

Fin
de la teoría 4

Práctica sobre la teoría 4

Práctica sobre la teoría 4

1) Definir una clase **Persona** con 3 campos públicos: **Nombre**, **Edad** y **DNI**. Escribir un algoritmo que permita al usuario ingresar en una consola una serie de datos de esta forma:

Nombre, Documento, Edad <ENTER>.

Una vez finalizada la entrada de datos, el programa debe imprimir en la consola un listado con 4 columnas de la siguiente forma:

Nro)	Nombre	Edad	DNI.
-------------	---------------	-------------	-------------

Ejemplo de listado por consola:

1)	Juan Perez	40	2098745
2)	José García	41	1965412
...			

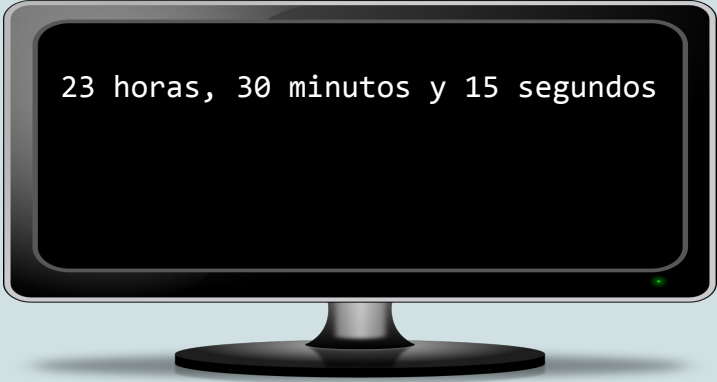
NOTA: Se puede utilizar: `Console.SetIn(new System.IO.StreamReader(nombreDeArchivo));` para cambiar la entrada estándar de la consola y poder leer directamente desde un archivo de texto.

2) Modificar el programa anterior haciendo privados todos los campos. Definir un constructor adecuado y un método público **Imprimir()** que escriba en la consola los campos del objeto con el formato requerido para el listado.

3) Agregar a la clase **Persona** un método **EsMayorQue(Persona p)** que devuelva verdadero si la persona que recibe el mensaje tiene más edad que la persona enviada como parámetro. Utilizarlo para realizar un programa que devuelva la persona más joven de la lista.

4) Codificar la clase **Hora** de tal forma que el siguiente código produzca la salida por consola que se observa.

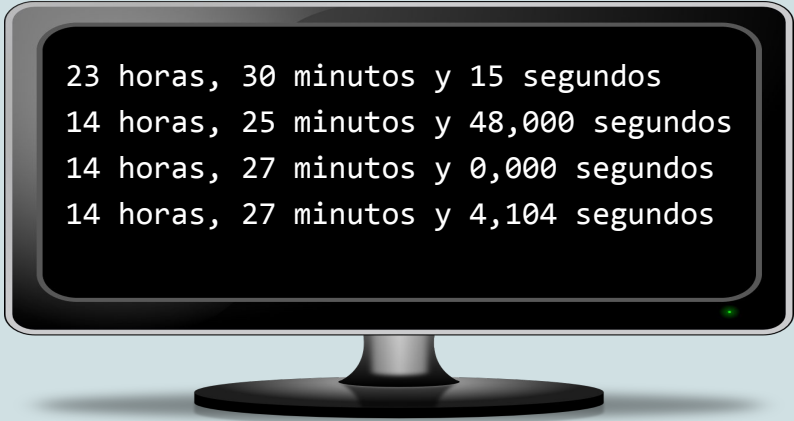
```
Hora h = new Hora(23,30,15);  
h.Imprimir
```



23 horas, 30 minutos y 15 segundos

5) Agregar un segundo constructor a la clase **Hora** para que pueda especificarse la hora por medio de un único valor que admita decimales. Por ejemplo 3,5 indica la hora 3 y 30 minutos. Si se utiliza este segundo constructor, el método imprimir debe mostrar los segundos con tres dígitos decimales. Así el siguiente código debe producir la salida por consola que se observa.

```
new Hora(23, 30, 15).Imprimir();  
new Hora(14.43).Imprimir();  
new Hora(14.45).Imprimir();  
new Hora(14.45114).Imprimir();
```



23 horas, 30 minutos y 15 segundos
14 horas, 25 minutos y 48,000 segundos
14 horas, 27 minutos y 0,000 segundos
14 horas, 27 minutos y 4,104 segundos

6) Codificar una clase llamada **Ecuacion2** para representar una ecuación de 2º grado. Esta clase debe tener 3 campos privados, los coeficientes a, b y c de tipo **double**. La única forma de establecer los valores de estos campos será en el momento de la instanciación de un objeto **Ecuacion2**.

Codificar los siguientes métodos:

- **GetDiscriminante()**: devuelve el valor del discriminante (**double**), el discriminante tiene la siguiente formula, $(b^2) - 4 * a * c$.
- **GetCantidadDeRaices()**: devuelve 0, 1 ó 2 dependiendo de la cantidad de raíces reales que posee la ecuación. Si el discriminante es negativo no tiene raíces reales, si el discriminante es cero tiene una única raíz, si el discriminante es mayor que cero posee 2 raíces reales.
- **ImprimirRaices()**: imprime la única o las 2 posibles raíces reales de la ecuación. En caso de no poseer soluciones reales debe imprimir una leyenda que así lo especifique.

7) Implementar la clase **Matriz** que se utilizará para trabajar con matrices matemáticas. Implementar los dos constructores y todos los métodos que se detallan a continuación:

```
public Matriz(int filas, int columnas)
public Matriz(double[,] matriz)
public void SetElemento(int fila, int columna, double elemento)
public double GetElemento(int fila, int columna)
public void imprimir()
public void imprimir(string formatString)
public double[] GetFila(int fila)
public double[] GetColumna(int columna)
public double[] GetDiagonalPrincipal()
public double[] GetDiagonalSecundaria()
public double[][] getArregloDeArreglo()
public void sumarle(Matriz m)
public void restarle(Matriz m)
public void multiplicarPor(Matriz m)
```

Práctica sobre la teoría 4

8) Prestar atención a los siguientes programas (ninguno funciona correctamente):

```
Foo f = new Foo();  
f.Imprimir();
```

```
class Foo  
{  
    string? _bar;  
    public void Imprimir()  
    {  
        Console.WriteLine(_bar.Length);  
    }  
}
```

```
Foo f = new Foo();  
f.Imprimir();
```

```
class Foo  
{  
    public void Imprimir()  
    {  
        string? bar;  
        Console.WriteLine(bar.Length);  
    }  
}
```

¿Qué se puede decir acerca de la inicialización de los objetos? ¿En qué casos son inicializados automáticamente y con qué valor?

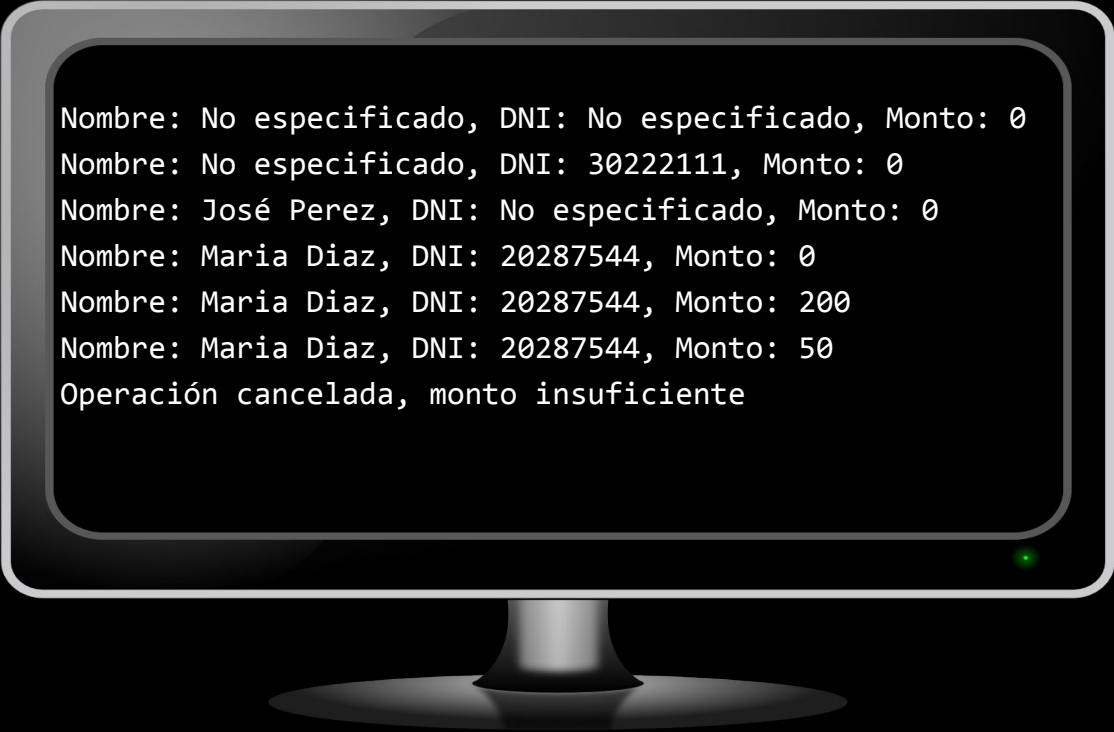
9) ¿Qué se puede decir en relación a la sobrecarga de métodos en este ejemplo?

```
class A  
{  
    public void Metodo(short n) {  
        Console.Write("short {0} - ", n);  
    }  
    public void Metodo(int n) {  
        Console.Write("int {0} - ", n);  
    }  
    public int Metodo(int n) {  
        return n + 10;  
    }  
}
```

10) Completar la clase **Cuenta** para que el siguiente código produzca la salida indicada:

```
Cuenta cuenta = new Cuenta();  
cuenta.Imprimir();  
cuenta = new Cuenta(30222111);  
cuenta.Imprimir();  
cuenta = new Cuenta("José Perez");  
cuenta.Imprimir();  
cuenta = new Cuenta("Maria Diaz", 20287544);  
cuenta.Imprimir();  
cuenta.Depositar(200);  
cuenta.Imprimir();  
cuenta.Extraer(150);  
cuenta.Imprimir();  
cuenta.Extraer(1500);
```

```
class Cuenta  
{  
    private double _monto;  
    private int _titularDNI;  
    private string? _titularNombre;  
    . . .  
}
```



```
Nombre: No especificado, DNI: No especificado, Monto: 0  
Nombre: No especificado, DNI: 30222111, Monto: 0  
Nombre: José Perez, DNI: No especificado, Monto: 0  
Nombre: Maria Diaz, DNI: 20287544, Monto: 0  
Nombre: Maria Diaz, DNI: 20287544, Monto: 200  
Nombre: Maria Diaz, DNI: 20287544, Monto: 50  
Operación cancelada, monto insuficiente
```

Utilizar en la medida de lo posible la sintaxis **:this** en el encabezado de los constructores para invocar a otro constructor ya definido.

11) Reemplazar estas líneas de código por otras equivalentes que utilicen el operador null-coalescing (??) y / o la asignación null-coalescing (??=)

```
...
if (st1 == null)
{
    if (st2 == null)
    {
        st = st3;
    }
    else
    {
        st = st2;
    }
}
else
{
    st = st1;
}
if (st4 == null)
{
    st4 = "valor por defecto";
}
...
```

Práctica sobre la teoría 4

12) Crear una solución con tres proyectos: una biblioteca de clases llamada **Automotores**, una biblioteca de clases llamada **Figuras** y una aplicación de consola llamada **Aplicacion**. La biblioteca **Automotores** debe contener una clase pública **Auto** (codificada de la misma manera que la vista en la teoría). La biblioteca **Figuras** debe contener las clases públicas **Circulo** y **Rectangulo**, codificadas de tal forma que el siguiente código (escrito en **Program.cs** del proyecto **Aplicacion**) produzca la siguiente salida

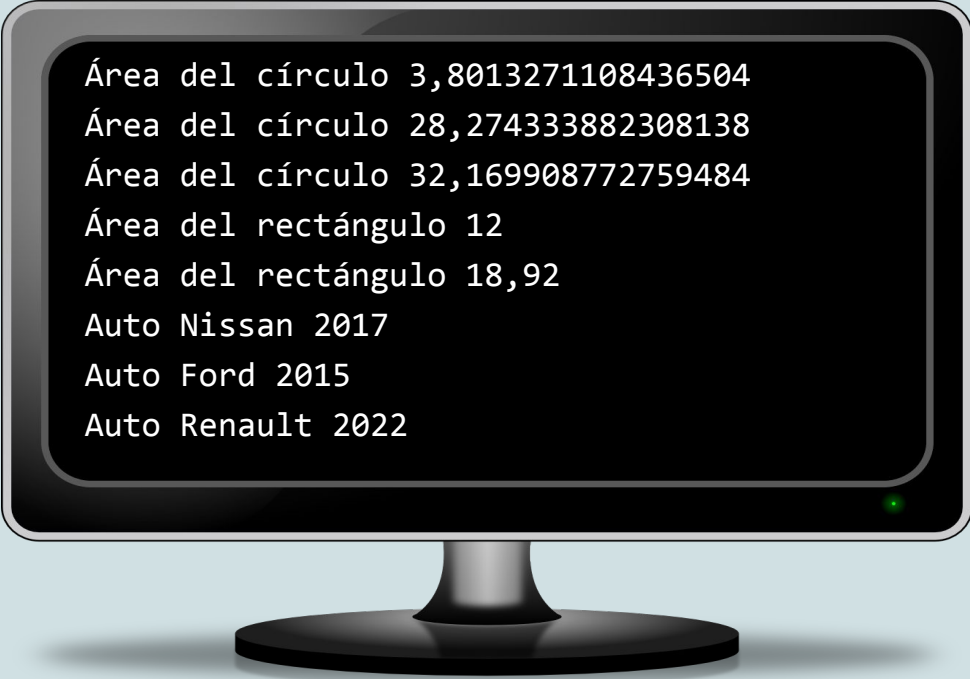
```
using Figuras;
using Automotores;

//El constructor de Circulo espera recibir el radio
List<Circulo> listaCirculos = [
    new Circulo(1.1),
    new Circulo(3),
    new Circulo(3.2)
];

//El constructor de Rectángulo espera recibir la base y la altura
List<Rectangulo> listaRectangulos = [
    new Rectangulo(3, 4),
    new Rectangulo(4.3, 4.4)
];

//La clase Auto está codificada como la vista en la teoría
List<Auto> listaAutos = [
    new Auto("Nissan", 2017),
    new Auto("Ford", 2015),
    new Auto("Renault")
];
```

```
foreach (Circulo c in listaCirculos)
{
    Console.WriteLine($"Área del círculo {c.GetArea()}");
}
foreach (Rectangulo r in listaRectangulos)
{
    Console.WriteLine($"Área del rectángulo {r.GetArea()}");
}
foreach (Auto a in listaAutos)
{
    Console.WriteLine(a.GetDescripcion());
}
```



Área del círculo 3,8013271108436504
Área del círculo 28,274333882308138
Área del círculo 32,169908772759484
Área del rectángulo 12
Área del rectángulo 18,92
Auto Nissan 2017
Auto Ford 2015
Auto Renault 2022