



# .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#

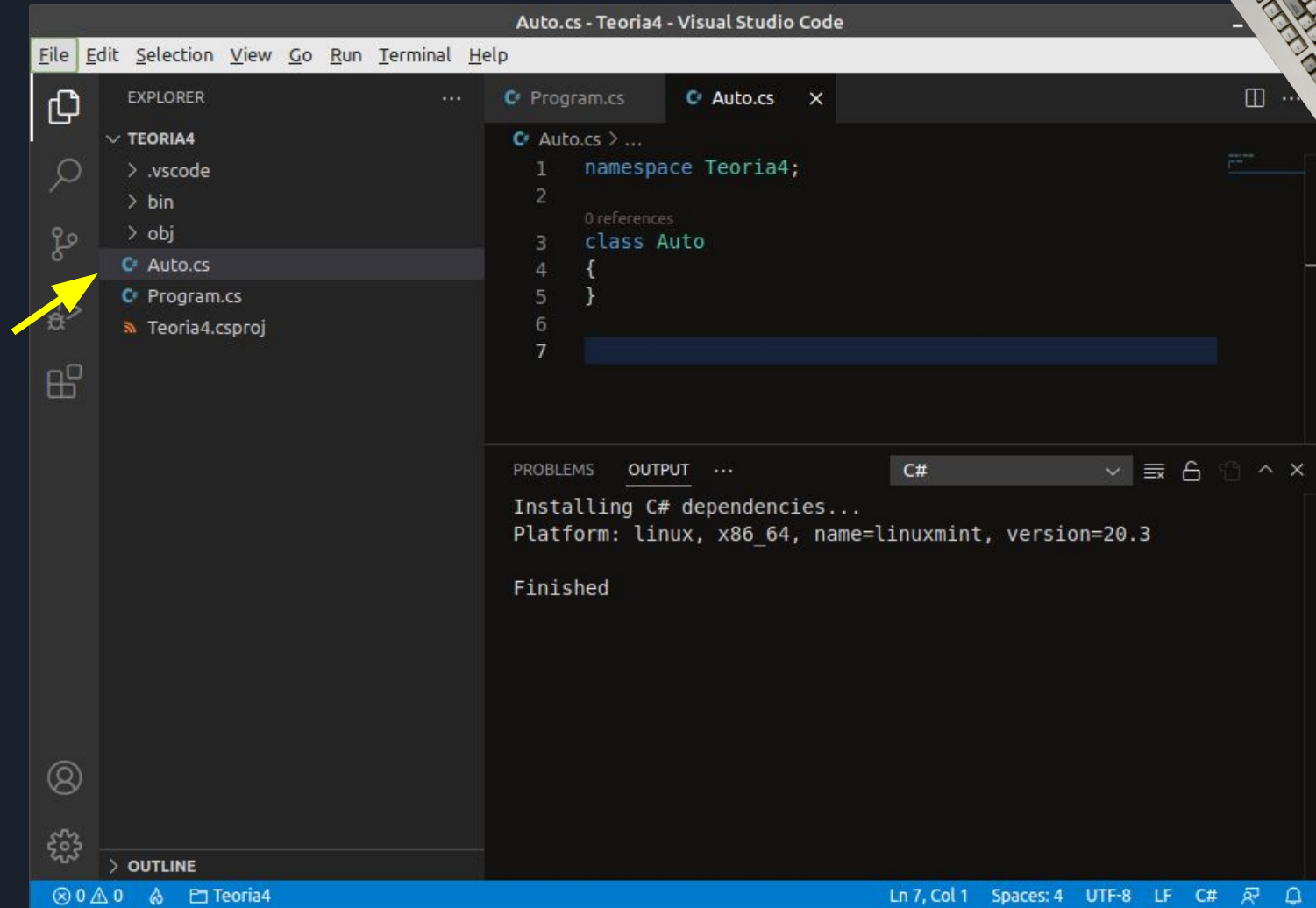
1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria4`
4. Abrir `Visual Studio Code` sobre este proyecto







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



## Auto.cs - Teoria4 - Visual Studio Code

al Help

Program.cs

Auto.cs X

Auto.cs &gt; ...

1 namespace Teoria4;

2

0 references

3 class Auto

4 {

5 }

6

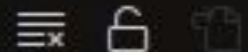
7

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`

PROBLEMS OUTPUT ...

C#



Installing C# dependencies...

Platform: linux, x86\_64, name=linuxmint, version=20.3



# 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();
```

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

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

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;
```

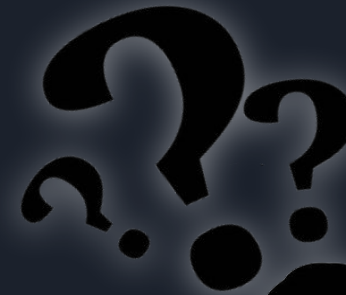
```
Console.WriteLine(a);
```





## 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";
```

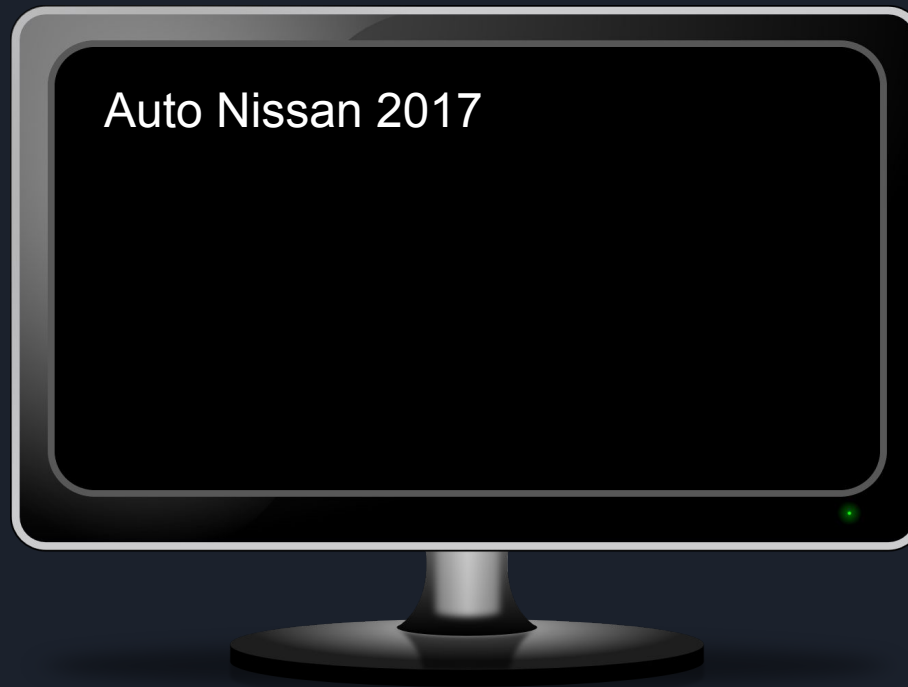
```
a.Modelo = 2017;
```

```
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}");
```



Auto Nissan 2017  
Auto Ford 2015

# 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
```

```
{
```

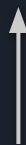
```
    → private string? _marca;
```

```
    → 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());
```





## 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());
```



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);
```

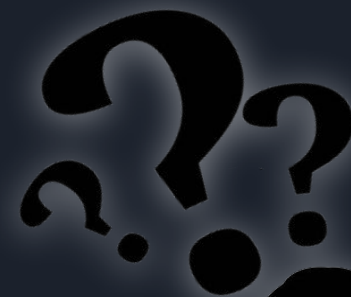
```
Console.WriteLine(a.ObtenerDescripcion());
```

```
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.



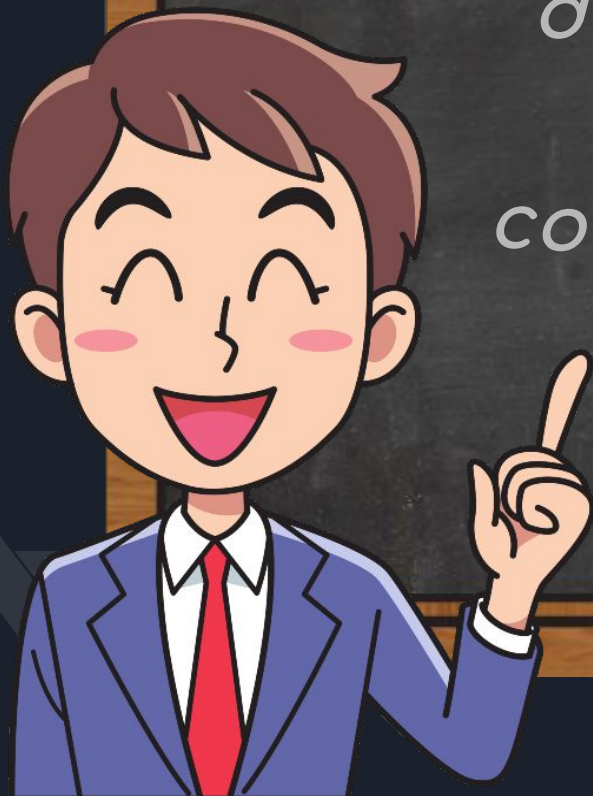
*Si se define un  
constructor  
explícitamente,  
el compilador  
NO lo sobrecarga con el  
constructor por defecto*

*¿ 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,
    {
        _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.

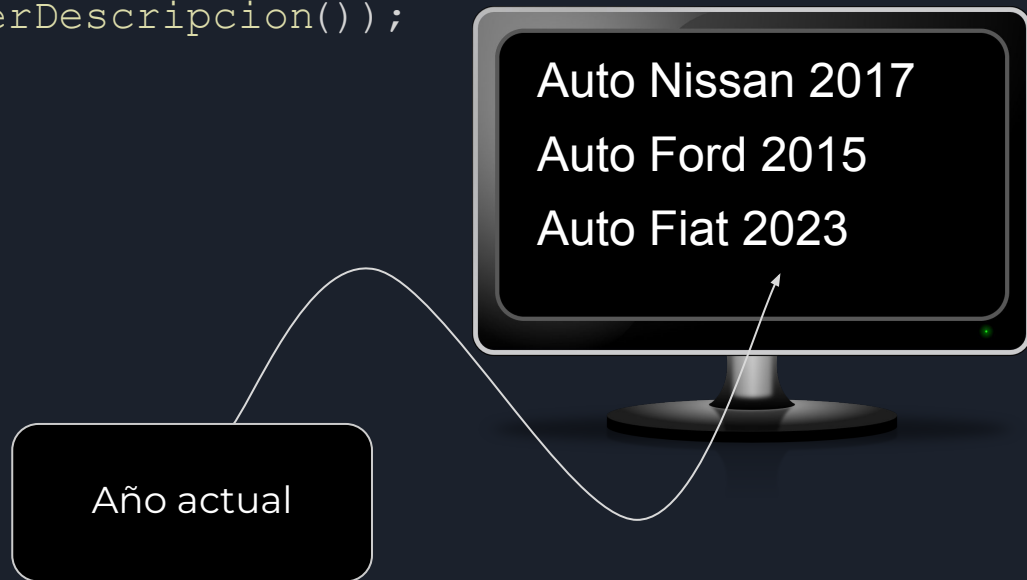
```
using Teoria4;

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 2023
```

## 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







## 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)
```

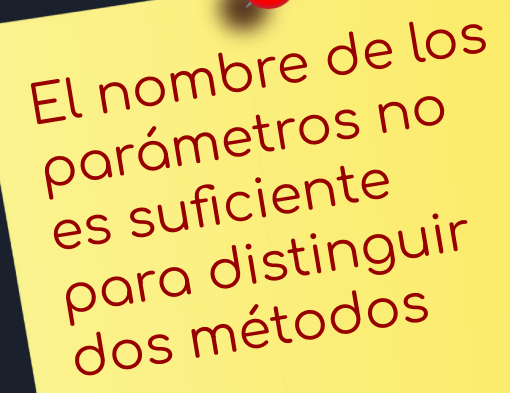
```
void procesar(out double valor)
```

## Métodos de instancia. Sobrecarga

### Ejemplos de sobrecargas inválidas

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

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



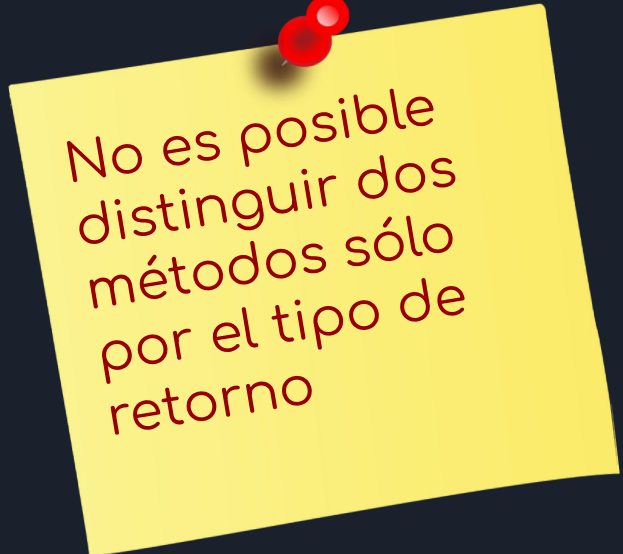
El nombre de los  
parámetros no  
es suficiente  
para distinguir  
dos métodos

## Métodos de instancia. Sobrecarga

### Ejemplos de sobrecargas inválidas

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

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



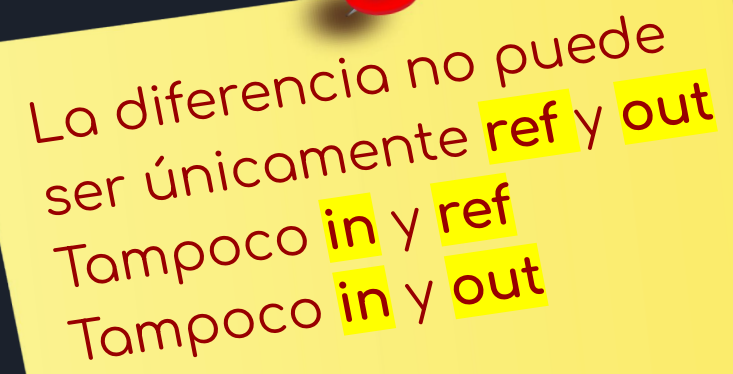
No es posible  
distinguir dos  
métodos sólo  
por el tipo de  
retorno

## Métodos de instancia. Sobrecarga

### Ejemplos de sobrecargas inválidas

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

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

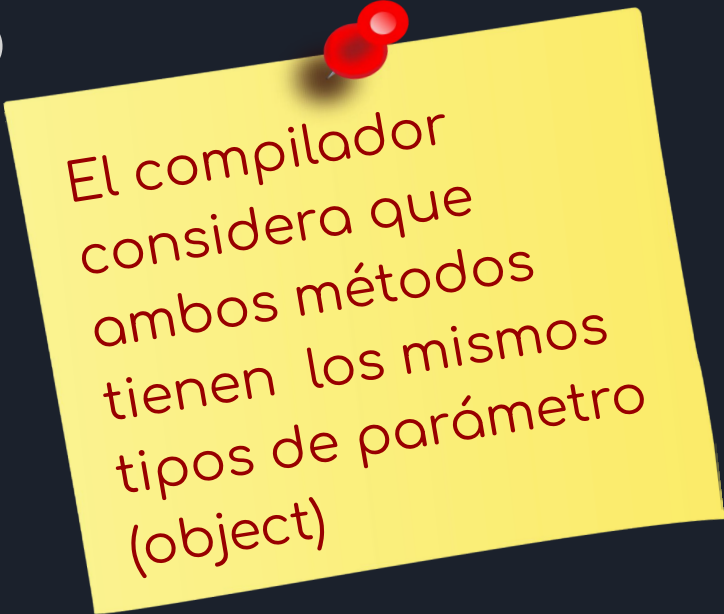


La diferencia no puede  
ser únicamente **ref** y **out**  
Tampoco **in** y **ref**  
Tampoco **in** y **out**

## 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);  
    }  
}
```

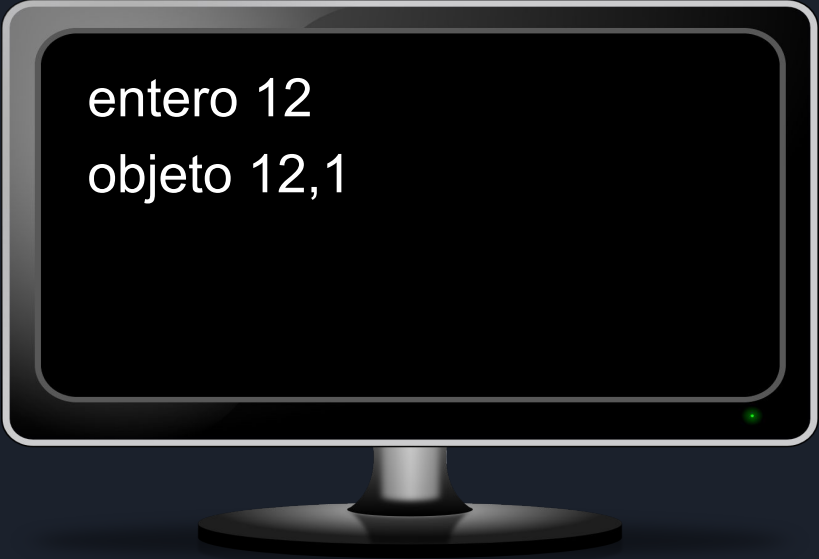
## Métodos de instancia - Sobrecarga

```
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



float 12

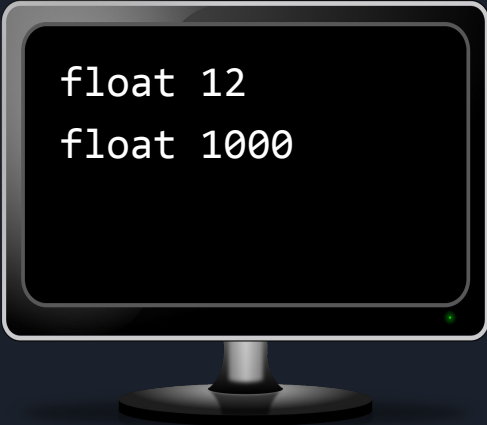
```
class Sobrecarga  
{  
    public void Procesar(object valor1)  
    {  
        Console.WriteLine("objeto " + valor1);  
    }  
    public void Procesar(float valor1)  
    {  
        Console.WriteLine("float " + valor1);  
    }  
}
```

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 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 monolítica y modular

- **Aplicación monolítica:** Único módulo ejecutable que implementamos en un único proyecto (lo que hicimos hasta ahora)
- **Aplicación 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**:

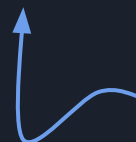
```
cd Estacionamiento
```

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
```

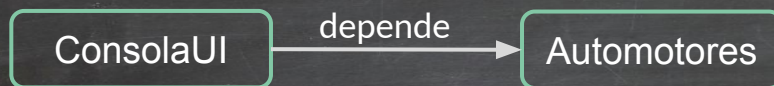
```
dotnet sln add ./ConsolaUI
```

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**:

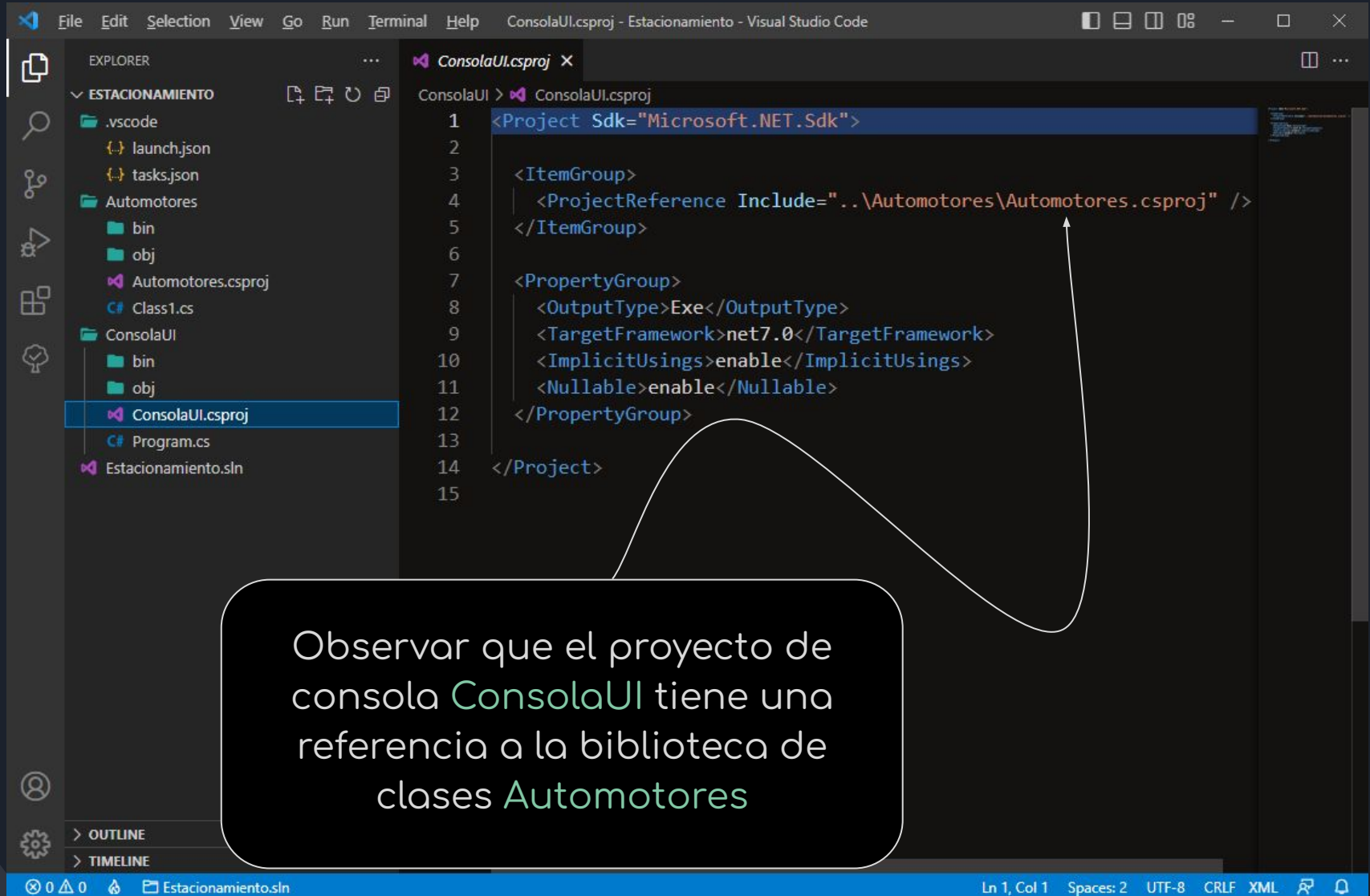
```
dotnet add ./ConsolaUI reference ./Automotores
```

Abrir Visual Studio Code

```
code .
```



# Arquitectura modular



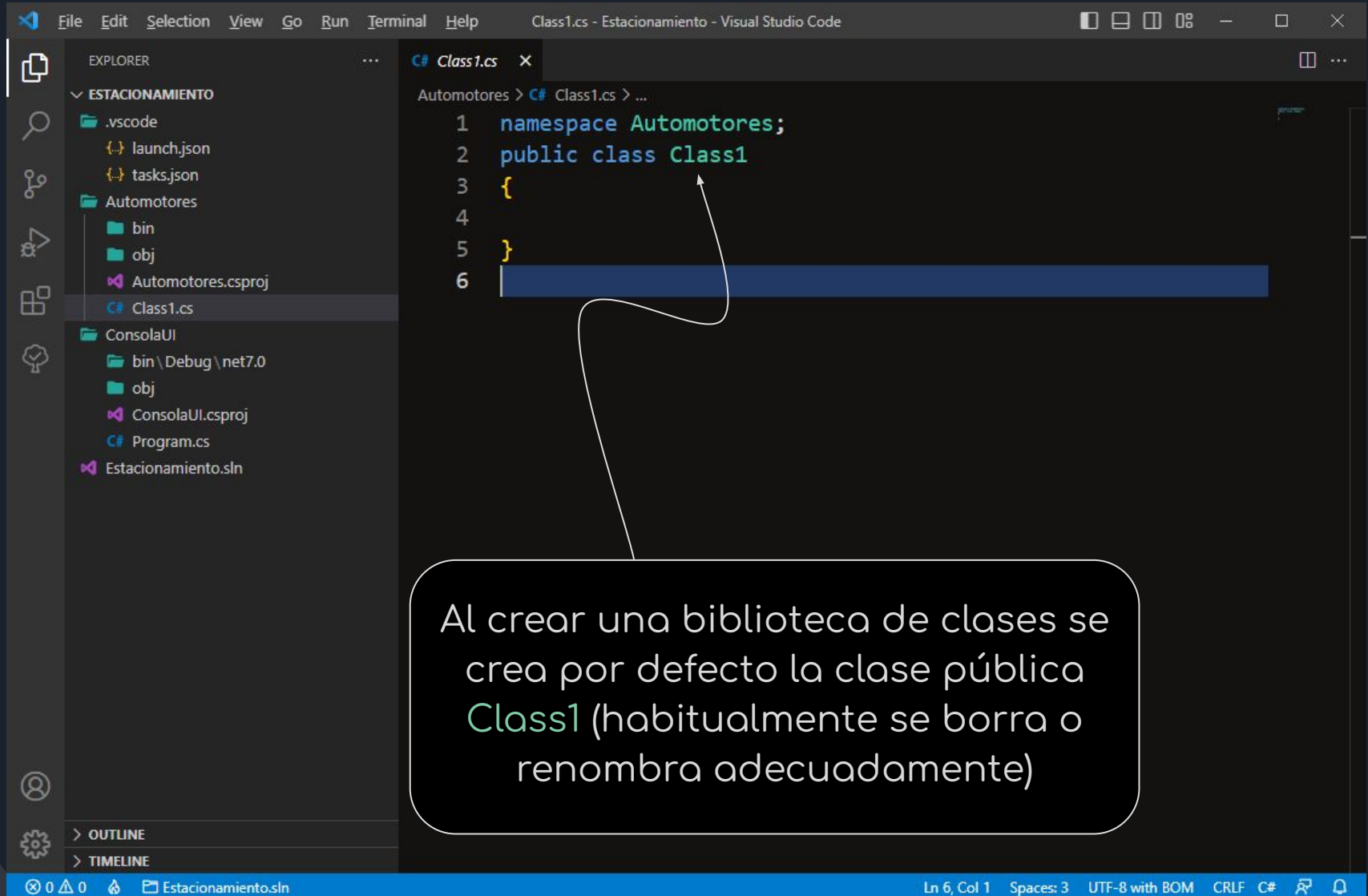
The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane displays a project named 'ESTACIONAMIENTO' with a folder 'ConsolaUI' containing 'ConsolaUI.csproj'. The main editor shows the content of 'ConsolaUI.csproj', which includes a project reference to '..\Automotores\Automotores.csproj'. A white callout box with a curved arrow points from the text inside to the 'Include' attribute in the XML. The status bar at the bottom indicates the file is 'Estacionamiento.sln' and the editor is at 'Ln 1, Col 1'.

```
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <ItemGroup>
4     <ProjectReference Include="..\Automotores\Automotores.csproj" />
5   </ItemGroup>
6
7   <PropertyGroup>
8     <OutputType>Exe</OutputType>
9     <TargetFramework>net7.0</TargetFramework>
10    <ImplicitUsings>enable</ImplicitUsings>
11    <Nullable>enable</Nullable>
12  </PropertyGroup>
13
14 </Project>
15
```

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



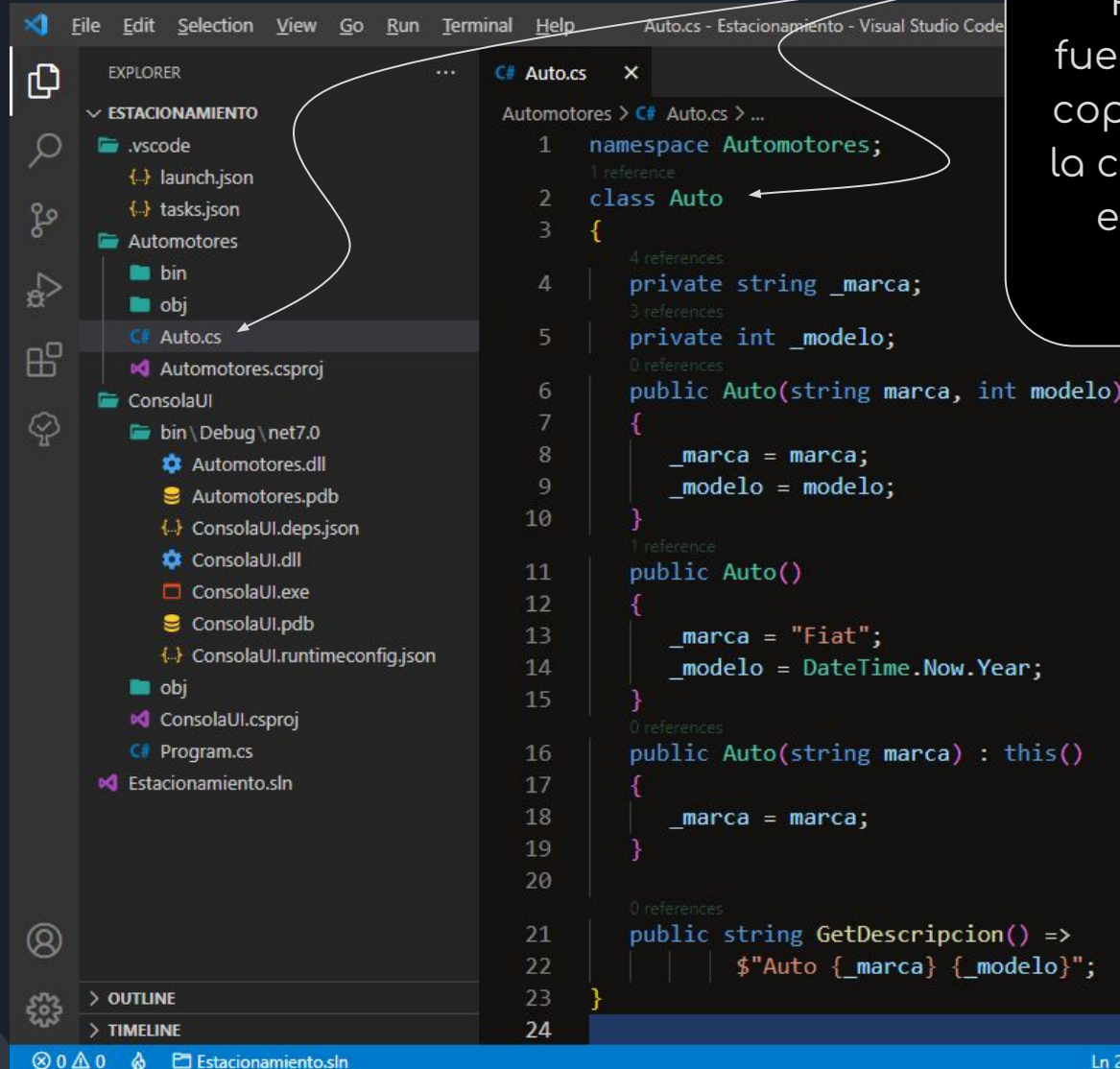
# Arquitectura modular





# Arquitectura modular

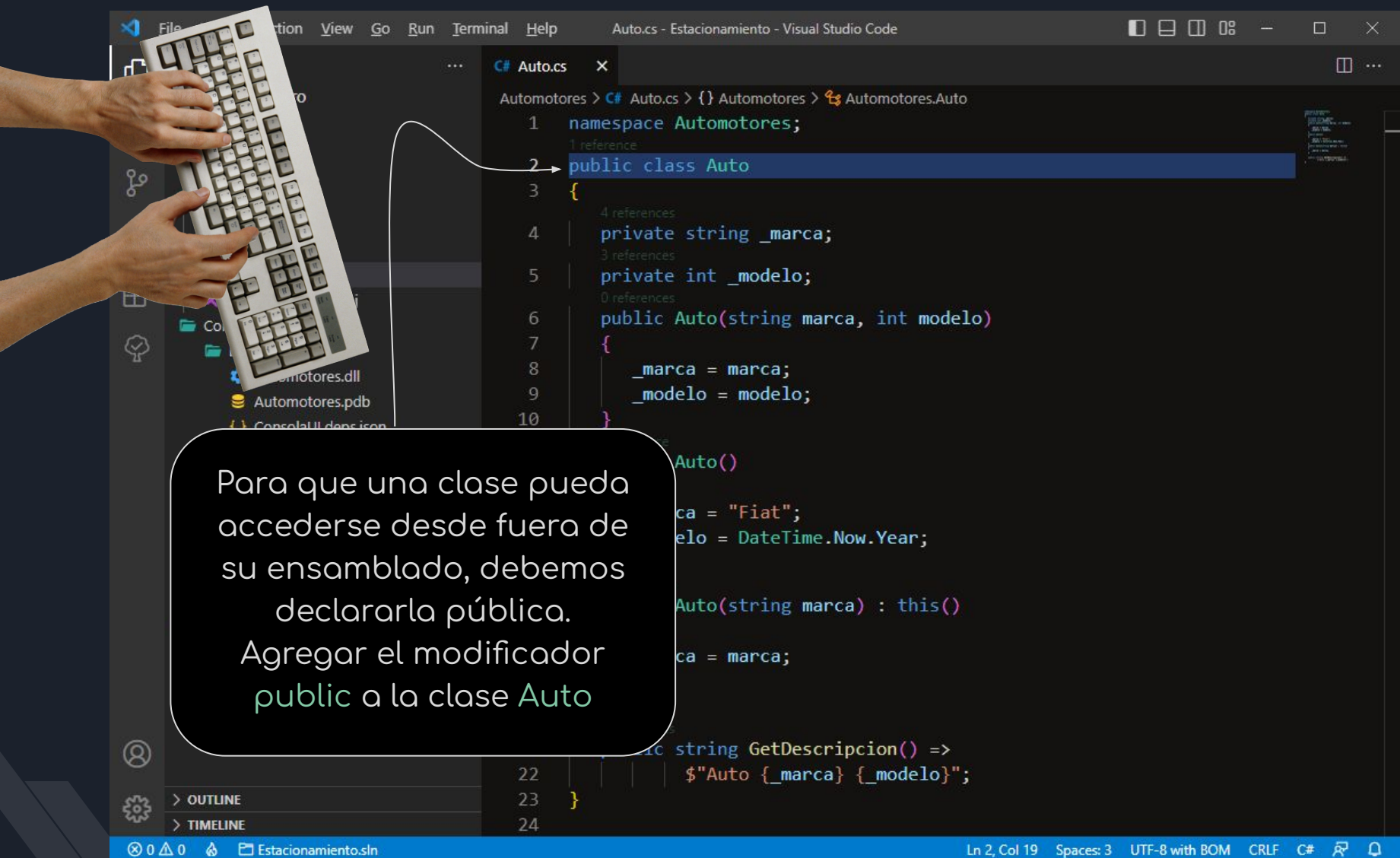
Renombrar el archivo fuente `Class1.cs` a `Auto.cs` y copiar en éste el código de la clase `Auto` que definimos en el proyecto anterior (Teoría4)



```
1 namespace Automotores;
2 class Auto
3 {
4     private string _marca;
5     private int _modelo;
6     public Auto(string marca, int modelo)
7     {
8         _marca = marca;
9         _modelo = modelo;
10    }
11    public Auto()
12    {
13        _marca = "Fiat";
14        _modelo = DateTime.Now.Year;
15    }
16    public Auto(string marca) : this()
17    {
18        _marca = marca;
19    }
20
21    public string GetDescripcion() =>
22        $"Auto {_marca} {_modelo}";
23 }
24
```



# Arquitectura modular



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

```
1 namespace Automotores;
2 public class Auto
3 {
4     private string _marca;
5     private int _modelo;
6     public Auto(string marca, int modelo)
7     {
8         _marca = marca;
9         _modelo = modelo;
10    }
11
12    Auto()
13    {
14        _marca = "Fiat";
15        _modelo = DateTime.Now.Year;
16    }
17
18    Auto(string marca) : this()
19    {
20        _marca = marca;
21    }
22
23    public string GetDescripcion() =>
24        $"Auto {_marca} {_modelo}";
25 }
```



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");
```

```
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

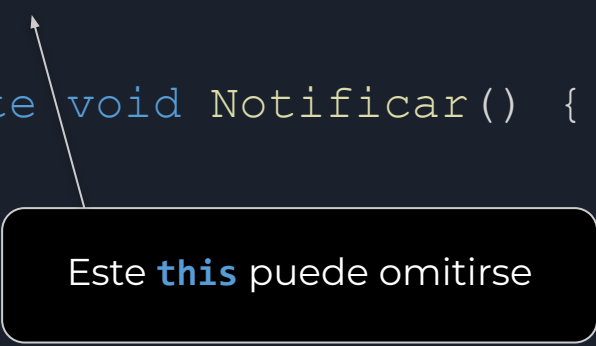




## Miembros de instancia. Uso de `this`

### 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