



.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 otros tipos de miembros que iremos viendo en este curso

La sintaxis para definir una clase ya la conocíamos. El comando `dotnet new console` crea el siguiente código

Una clase con un miembro

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```





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





Codificar la clase Auto



```
using System;
namespace Teoria4
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

```
class Auto
{
}
```

```
}
```



Codificar la clase Auto

```
using System;  
namespace Teoria4  
{
```

```
class Program  
{  
    static void Main(string[] args)  
    {  
    }  
}
```

Clase `Program`
con un miembro,
el método `Main`

```
class Auto  
{  
}
```

Clase `Auto` sin
ningún
miembro

```
}
```

Este código compila
y se ejecuta
correctamente,
aunque no hace
nada interesante



Agregar la línea resaltada



```
using System;
namespace Teoria4
{
    class Program
    {
        static void Main(string[] args)
        {
            Auto a;
        }
    }

    class Auto
    {
    }
}
```

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



Agregar las líneas resaltadas y ejecutar



```
using System;  
namespace Teoria4  
{
```

```
    class Program  
    {
```

```
        static void Main(string[] args)  
        {
```

```
            Auto a;
```

```
            a = new Auto();  
            Console.WriteLine(a);
```

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

Se imprime el objeto
en la consola

```
        }  
    }
```

```
class Auto  
{  
}
```

```
using System;
namespace Teoria4
{
    class Program
    {
        static void Main(string[] args)
        {
            Auto a;
            a = new Auto();
            Console.WriteLine(a);
        }
    }
    class Auto
    {
    }
}
```

`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 el método Main de la clase Program

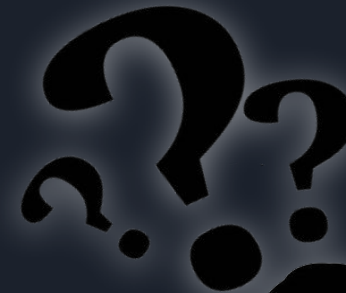


```
static void Main(string[] args)
{
    Auto a;
    a = new Auto();
    a.Marca = "Nissan";
    a.Modelo = 2017;
    Console.WriteLine(a);
}
```



Modificar el método Main de la clase Program

```
static void Main(string[] args)
{
    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 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 cualquier método miembro 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;
}
```





Modificar el método Main de la clase Program

```
static void Main(string[] args)
{
```

```
    Auto a;
```

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

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

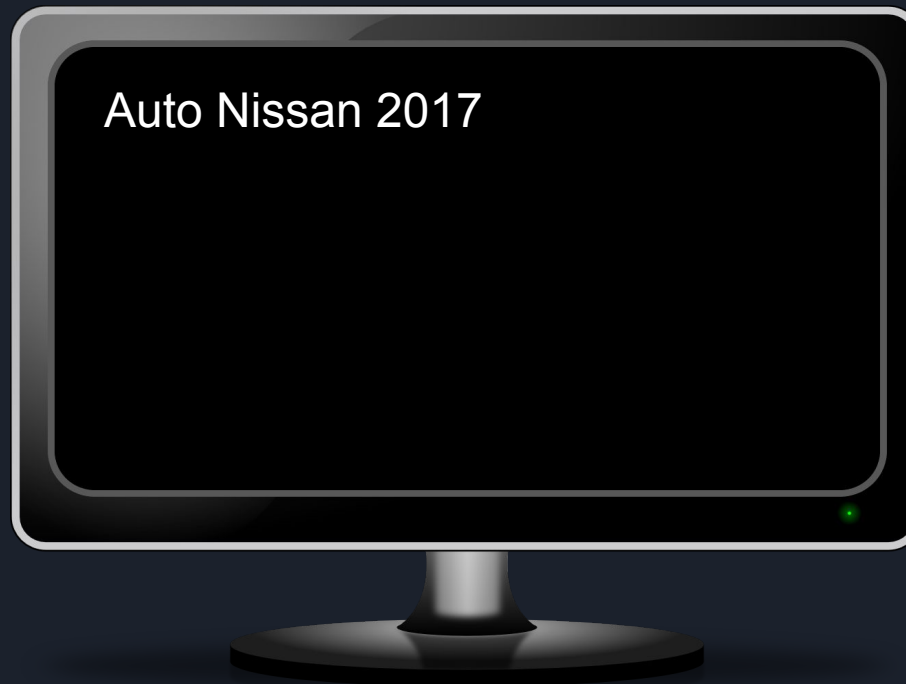
```
    a.Modelo = 2017;
```

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

```
}
```



```
static void Main(string[] args)
{
    Auto a;
    a = new Auto();
    a.Marca = "Nissan";
    a.Modelo = 2017;
    Console.WriteLine($"Auto {a.Marca} {a.Modelo}");
}
```





Agregar las siguientes líneas:

```
static void Main(string[] args)
{
    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}");
}
```



```
static void Main(string[] args)
{
    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 Imprimir() en la clase Auto para que los objetos autos sean responsables de imprimirse a sí mismo.



```
class Auto
{
    public string Marca;
    public int Modelo;
    public void Imprimir()
    {
        Console.WriteLine($"Auto {Marca} {Modelo}");
    }
}
```



Implementar el método Imprimir() en la clase Auto para que los objetos autos sean responsables de imprimirse a sí mismo.

```
class Auto
{
    public string Marca;
    public int Modelo;
    public void Imprimir() =>
        Console.WriteLine($"Auto {Marca} {Modelo}");
}
```

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

Métodos de instancia

El método `Imprimir()` de la clase `Auto` es un método de instancia (no lleva la palabra clave `static`)

```
public void Imprimir()
```

A diferencia del método `Main()` que es un método estático de la clase `Program`

```
static void Main(string[] args)
```





Modificar Main



```
static void Main(string[] args)
{
    Auto a;
    a = new Auto();
    a.Marca = "Nissan";
    a.Modelo = 2017;
    a.Imprimir();
    Auto b = new Auto();
    b.Modelo = 2015;
    b.Marca = "Ford";
    b.Imprimir();
}
```

```
static void Main(string[] args)
{
    Auto a;
    a = new Auto();
    a.Marca = "Nissan";
    a.Modelo = 2017;
    a.Imprimir();
    Auto b = new Auto();
    b.Modelo = 2015;
    b.Marca = "Ford";
    b.Imprimir();
}
```

Le dimos la responsabilidad de imprimirse a los propios autos. Así evitamos tener que acceder a la representación interna de los de los autos al momento de 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 void Imprimir() =>  
        Console.WriteLine($"Auto {_marca} {_modelo}");  
}
```





Modificar la clase Auto. Hacer privados sus campos

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

    public void Imprimir() =>
        Console.WriteLine($"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 void Imprimir() =>
        Console.WriteLine($"Auto {_marca} {_modelo}");
}
```





Modificar el método Main de la la clase Program y ejecutar.

```
static void Main(string[] args)
{
```

```
    Auto a;
```

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

```
    a.Imprimir();
```

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

```
    b.Imprimir();
```

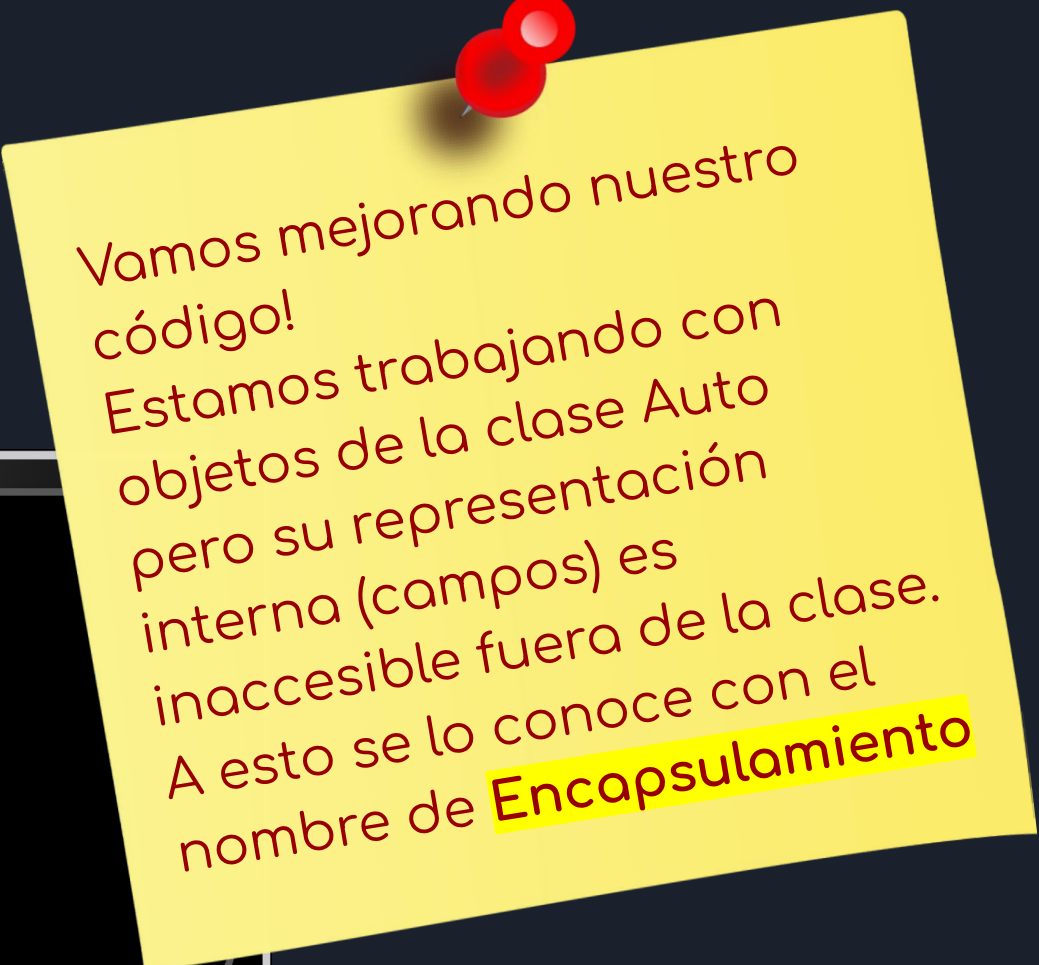
```
}
```



```
static void Main(string[] args)
{
    Auto a;
    a = new Auto("Nissan", 2017);
    a.Imprimir();
    Auto b = new Auto("Ford", 2015);
    b.Imprimir();
}
```



Auto Nissan 2017
Auto Ford 2015

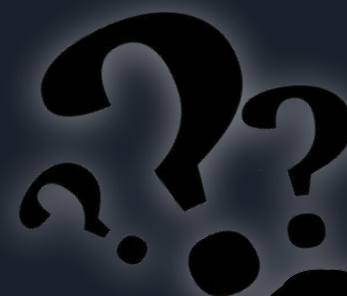


Vamos mejorando nuestro código!
Estamos trabajando con objetos de la clase Auto pero su representación interna (campos) es inaccesible fuera de la clase. A esto se lo conoce con el nombre de **Encapsulamiento**



Agregar la línea resaltada

```
static void Main(string[] args)
{
    Auto a;
    a = new Auto("Nissan", 2017);
    a.Imprimir();
    Auto b = new Auto("Ford", 2015);
    b.Imprimir();
    a = new Auto();
}
```



¿Cuál es el problema ?



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.



Constructores de instancia

Agregar la línea resaltada

```
static void Main(string[] args)
{
    Auto a;
    a = new Auto("Nissan", 2017);
    a.Imprimir();
    Auto b = new Auto("Ford", 2015);
    b.Imprimir();
    a = new Auto();
}
```



Tenemos un error
pero podemos
resolverlo agregando
explícitamente el
constructor por
defecto a la clase
Auto



Agregar el constructor que el compilador ya no incluye por nosotros



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




Agregar el constructor que el compilador ya no incluye por nosotros

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

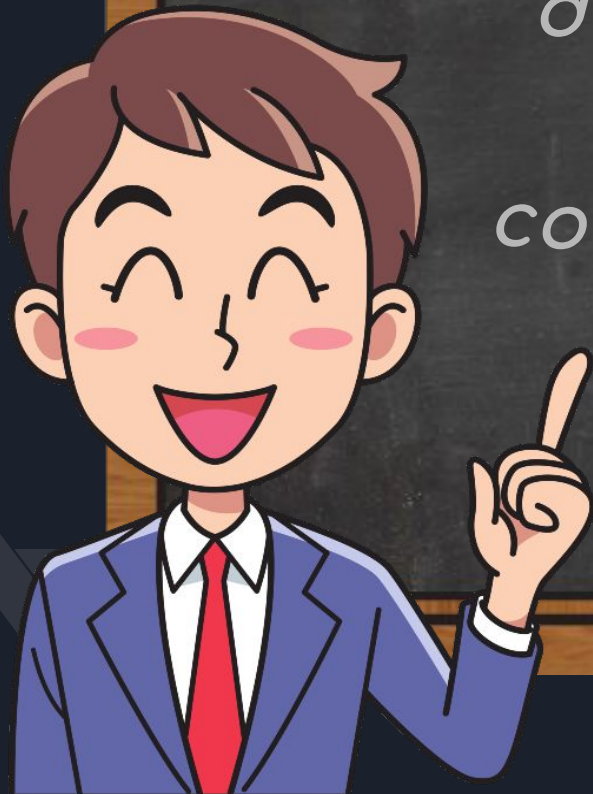
Acabamos de **sobrecargar** al constructor definiendo dos versiones distintas. Ahora compila sin errores.

*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.*



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



Modificar el constructor adecuado para obtener la siguiente salida

```
static void Main(string[] args)
{
    Auto a;
    a = new Auto("Nissan", 2017);
    a.Imprimir();
    Auto b = new Auto("Ford", 2015);
    b.Imprimir();
    a = new Auto();
    a.Imprimir();
}
```



```
Auto Nissan 2017
Auto Ford 2015
Auto Fiat 2021
```

Año actual

Constructores de instancia. Sobrecarga

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

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

    . . .
}
```





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

```
static void Main(string[] args)
{
    Auto a;
    a = new Auto("Nissan", 2017);
    a.Imprimir();
    Auto b = new Auto("Ford", 2015);
    b.Imprimir();
    a = new Auto("Renault");
    a.Imprimir();
}
```



```
Auto Nissan 2017
Auto Ford 2015
Auto Renault 2021
```

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)
```

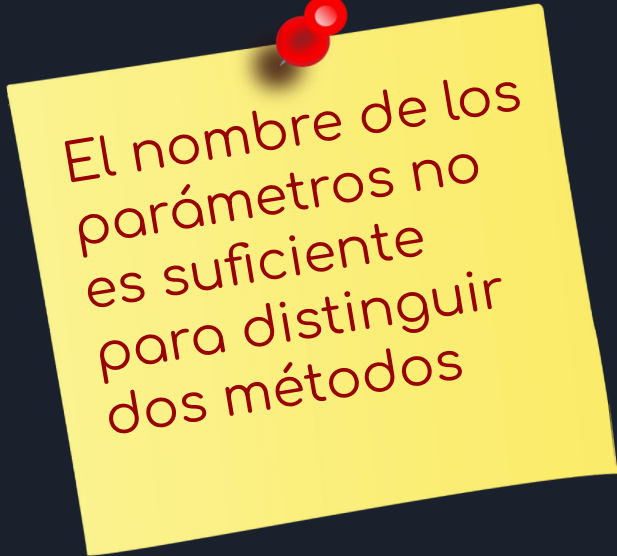
```
void procesar(out double valor)
```

Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

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

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



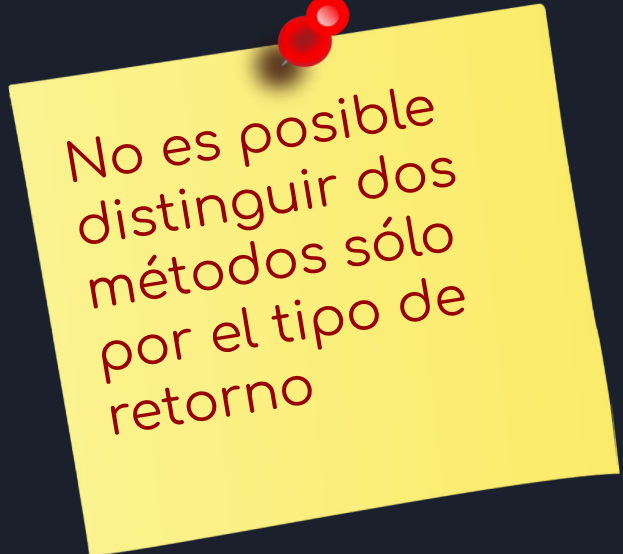
El nombre de los
parámetros no
es suficiente
para distinguir
dos métodos

Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

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

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



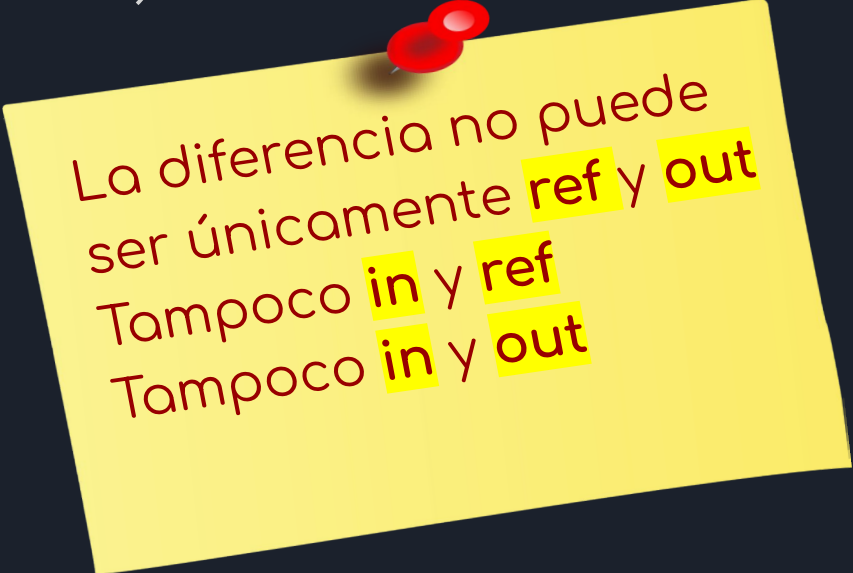
No es posible
distinguir dos
métodos sólo
por el tipo de
retorno

Métodos de instancia. Sobrecarga

Ejemplos de sobrecargas inválidas

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

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

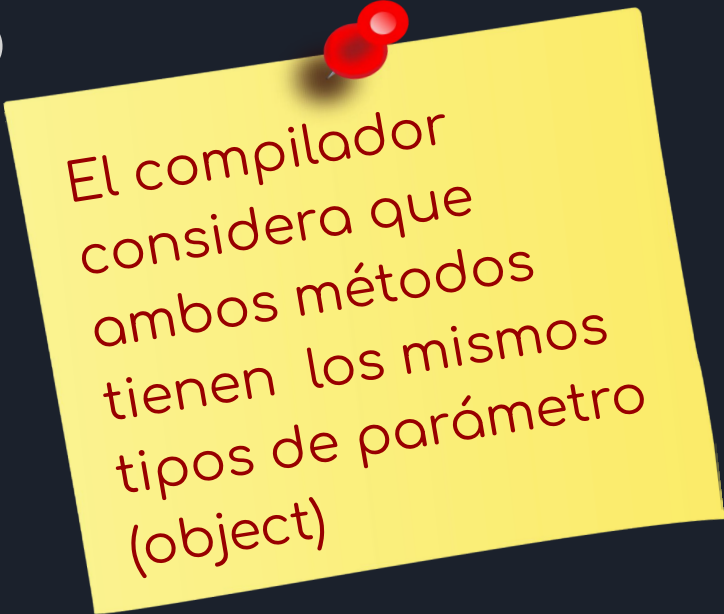


La diferencia no puede
ser únicamente **ref** y **out**
Tampoco **in** y **ref**
Tampoco **in** y **out**

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?

```
static void Main()
{
    procesar(12);
    procesar(12.1);
}
static void procesar(object valor1)
{
    Console.WriteLine("objeto " + valor1);
}
static void procesar(int valor1)
{
    Console.WriteLine("entero " + valor1);
}
```



```
static void Main()
{
    procesar(12);
    procesar(12.1);
}
static void procesar(object valor1)
{
    Console.WriteLine("objeto " + valor1);
}
static void procesar(int valor1)
{
    Console.WriteLine("entero " + valor1);
}
```



entero 12
objeto 12,1

Para pensar
¿Cuál es la salida por consola?

```
static void Main() {  
    object o = 12;  
    procesar(o);  
    o = 12.1;  
    procesar(o);  
}  
static void procesar(object valor1)  
{  
    Console.WriteLine("objeto " + valor1);  
}  
static void procesar(int valor1)  
{  
    Console.WriteLine("entero " + valor1);  
}
```



```
static void Main() {  
    object o = 12;  
    procesar(o);  
    o = 12.1;  
    procesar(o);  
}  
static void procesar(object valor1)  
{  
    Console.WriteLine("objeto " + valor1);  
}  
static void procesar(int valor1)  
{  
    Console.WriteLine("entero " + valor1);  
}
```

La resolución de la sobrecarga se realiza en tiempo de compilación



objeto 12
objeto 12,1

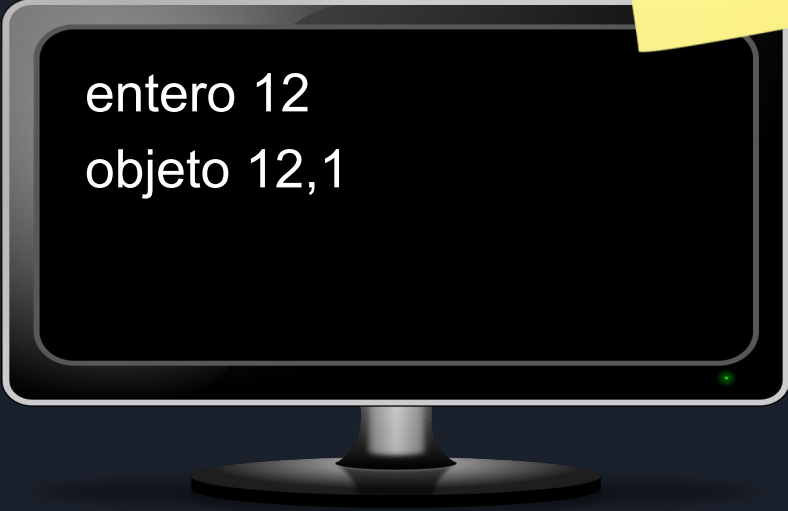
Para pensar
¿Cuál es la salida por consola?

```
static void Main() {  
    dynamic o = 12;  
    procesar(o);  
    o = 12.1;  
    procesar(o);  
}  
static void procesar(object valor1)  
{  
    Console.WriteLine("objeto " + valor1);  
}  
static void procesar(int valor1)  
{  
    Console.WriteLine("entero " + valor1);  
}
```



```
static void Main() {  
    dynamic o = 12;  
    procesar(o);  
    o = 12.1;  
    procesar(o);  
}  
static void procesar(object valor1)  
{  
    Console.WriteLine("objeto " + valor1);  
}  
static 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



Agregar las siguientes sobrecarga del método Imprimir() en la clase Auto

```
public void Imprimir() =>
    Console.WriteLine($"Auto {Marca} {Modelo}");
```

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

```
public void Imprimir(int repetir) {
    for (int i = 0; i < repetir; i++)
    {
        Imprimir();
    }
}
```

. . .





Modificar Main() y ejecutar

. . .

```
static void Main(string[] args)
{
    Auto a;
    a = new Auto("Nissan", 2017);
    a.Imprimir("IMPRESION");
    Auto b = new Auto("Ford", 2015);
    b.Imprimir(2);
    a = new Auto("Renault");
    a.Imprimir();
}
```

. . .



```
IMPRESION
Auto Nissan 2017
Auto Ford 2015
Auto Ford 2015
Auto Renault 2021
```

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#");  
    "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 COMPILACION. **Length** no es un método, es una propiedad de la clase string, no puede utilizarse como si fuese una instrucción

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

Nota sobre invocación a métodos y constructores

Nuestro método **Main** que sólo instancia autos para imprimirlos puede simplificarse así:

```
static void Main(string[] args)
{
    new Auto("Nissan", 2017).Imprimir("IMPRESION");
    new Auto("Ford", 2015).Imprimir(2);
    new Auto("Renault").Imprimir();
}
```



```
IMPRESION
Auto Nissan 2017
Auto Ford 2015
Auto Ford 2015
Auto Renault 2021
```

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`

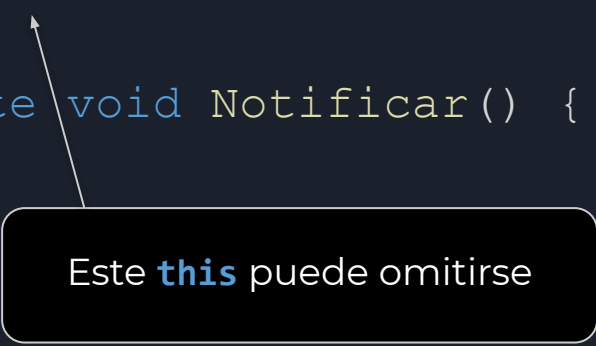
Dentro de un `constructor` o `método de instancia`, la palabra clave `this` hace referencia a la instancia (el propio objeto) que está ejecutando ese código.

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;
```

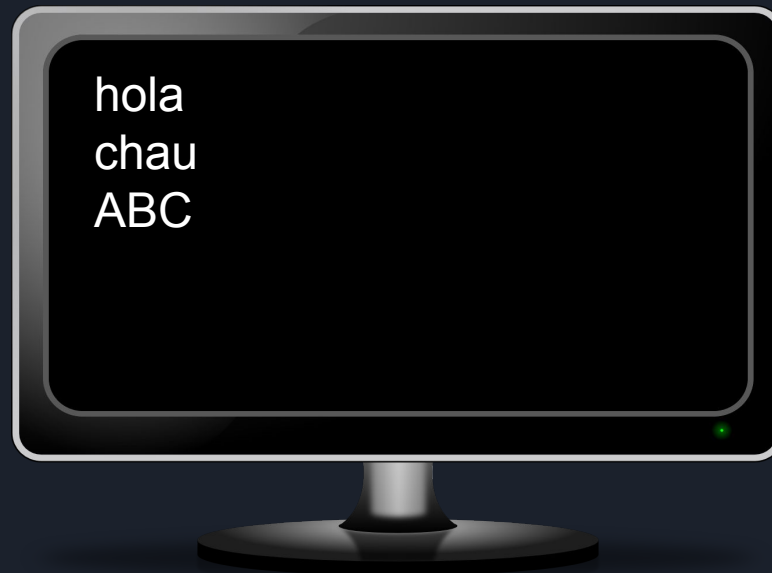
equivale a

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



```
static void Main()
{
    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



??= Asignación null-coalescing

(disponible a partir de C# 8.0)

a = a ?? b;

equivale a

a ??= b;



Fin