



# .NET

## Teoría 6

# Derivación de clases y herencia

# Herencia

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

# Sistema unificado de tipos





Vamos a presentar el concepto por medio de un ejemplo



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria6`
4. Abrir `Visual Studio Code` sobre este proyecto



# Codificar el enumerativo TipoAuto



```
namespace Teoria6;
```

```
enum TipoAuto
```

```
{
```

```
    Familiar,
```

```
    Deportivo,
```

```
    Camioneta
```

```
}
```



## Codificar la clase Auto

```
namespace Teoria6;
```

```
class Auto
```

```
{
```

```
    public string Marca = "";
```

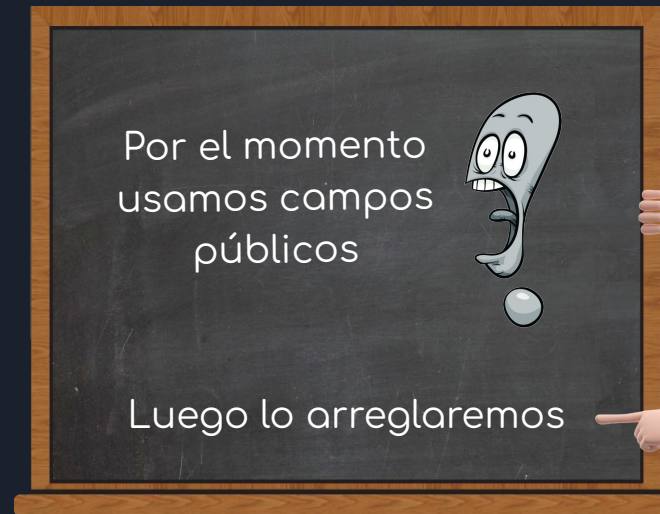
```
    public int Modelo;
```

```
    public TipoAuto Tipo;
```

```
    public void Imprimir()
```

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

```
}
```





# Codificar la clase Colectivo



```
namespace Teoria6;

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





# Codificar Program.cs de la siguiente manera y ejecutar



```
using Teoria6;

Auto a = new Auto();
Colectivo c = new Colectivo();
a.Marca = "Ford";
a.Modelo = 2000;
c.Marca = "Mercedes";
c.Modelo = 2010;
c.CantPasajeros = 20;
a.Tipo = TipoAuto.Deportivo;
a.Imprimir();
c.Imprimir();
```

# Derivación de clases y herencia

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

```
Auto a = new Auto();  
Colectivo c = new Colectivo();  
a.Marca = "Ford";  
a.Modelo = 2000;  
c.Marca = "Mercedes";  
c.Modelo = 2010;  
c.CantPasajeros = 20;  
a.Tipo = TipoAuto.Deportivo;  
a.Imprimir();  
c.Imprimir();
```

----- TipoAuto.cs -----

```
enum TipoAuto { Familiar, Deportivo, Camioneta }
```

----- Auto.cs -----

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

----- Colectivo.cs -----

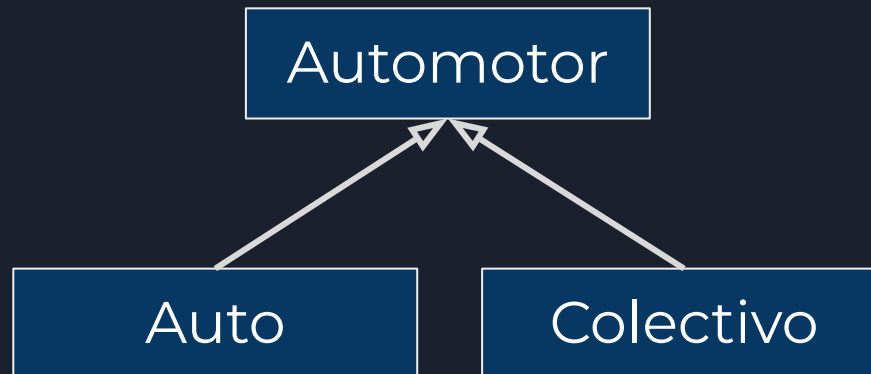
```
class Colectivo  
{  
    public string Marca = "";  
    public int Modelo;  
    public int CantPasajeros;  
    public void Imprimir()  
        => Console.WriteLine($"{Marca} {Modelo}");  
}
```



Ford 2000  
Mercedes 2010

# Derivación de clases

- Claramente las clases **Auto** y **Colectivo** comparten tanto atributos como comportamiento.
- Es posible por lo tanto generalizar el diseño colocando las características comunes en una superclase que llamaremos **Automotor**



Las flechas denotan una relación "es un"



Agregar la clase Automotor con los miembros comunes: Marca, Modelo e Imprimir()



```
namespace Teoria6;  
  
class Automotor {  
    public string Marca = "";  
    public int Modelo;  
    public void Imprimir()  
        => Console.WriteLine($"{Marca} {Modelo}");  
}
```



Derivar Auto y Colectivo de Automotor, borrar los miembros Marca Modelo e Imprimir() de ambas clases. Ejecutar



```
namespace Teoria6;
```

```
class Auto: Automotor  
{  
    public TipoAuto Tipo;  
}
```

Auto “deriva” de  
Automotor

---

```
namespace Teoria6;
```

```
class Colectivo: Automotor  
{  
    public int CantPasajeros;  
}
```

Auto “deriva” de  
Automotor

# Derivación de clases y herencia

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

```
Auto a = new Auto();  
Colectivo c = new Colectivo();  
a.Marca = "Ford";  
a.Modelo = 2000;  
c.Marca = "Mercedes";  
c.Modelo = 2010;  
c.CantPasajeros = 20;  
a.Tipo = TipoAuto.Deportivo;  
a.Imprimir();  
c.Imprimir();
```

----- TipoAuto.cs -----

```
enum TipoAuto { Familiar, Deportivo, Camioneta }
```

----- Automotor.cs -----

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

----- Auto.cs -----

```
class Auto : Automotor {  
    public TipoAuto Tipo;  
}
```

----- Colectivo.cs -----

```
class Colectivo : Automotor {  
    public int CantPasajeros;  
}
```



Ford 2000  
Mercedes 2010

# Derivación de clases

- Una clase derivada obtiene implícitamente **todos los miembros de la clase base, salvo sus constructores y sus finalizadores** (más adelante, en la sección “Para conocer más”).
- Las clases **Auto** y **Colectivo** del ejemplo derivan de la clase **Automotor**, por lo tanto un **Auto** es un **Automotor** y un **Colectivo** también es un **Automotor**.

# Derivación de clases

## Herencia de campos

- Los campos `Marca` y `Modelo` definidos en la clase `Automotor`, son heredados por las clases `Auto` y `Colectivo`, y por ello son válidas las siguientes asignaciones :

```
Auto a = new Auto();  
Colectivo c = new Colectivo();  
a.Marca = "Ford";  
a.Modelo = 2000;  
c.Marca = "Mercedes";  
c.Modelo = 2010;
```



NOTA: Todas las clases (menos **object**) en la plataforma .NET derivan directa o indirectamente de la clase **object**

```
class Automotor
```

```
{
```

```
...
```

```
}
```

=


```
class Automotor : object
```

```
{
```

```
...
```

```
}
```





## Derivación de clases

### Herencia de métodos

- Las clases `Auto` y `Colectivo` también heredan el método `Imprimir()` definido en `Automotor`.
- Sin embargo, el método `Imprimir()` resulta poco útil al no poder acceder a las variables específicas de `Auto` y `Colectivo` (`Tipo` y `CantPasajeros` respectivamente)
- Solución: Sobrescribir (`invalidar`) el método `Imprimir()` en cada una de las subclases para que tanto autos como colectivos se impriman de forma más adecuada.

Las clases derivadas pueden **invalidar** los métodos heredados para proporcionar una **implementación alternativa**.

Para poder invalidar un método, el método de la clase base debe marcarse con la palabra clave **virtual**.





## Hacer virtual al método Imprimir() de la clase Automotor:



```
namespace Teoria6;
```

```
class Automotor {
```

```
    public string Marca = "";
```

```
    public int Modelo;
```

```
    public virtual void Imprimir()
```

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

```
}
```

Anteponer el  
modificador  
virtual



## Codificar el método Imprimir() en la clase Colectivo y ejecutar



```
namespace Teoria6;  
  
class Colectivo : Automotor  
{  
    public int CantPasajeros;
```

El modificador **override** indica que se está invalidando el método heredado. Si no lo especificamos estaríamos ocultando el método en lugar de invalidarlo

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

Un método **override**, al igual que **virtual**, también puede ser invalidado en una clase derivada

Código en el archivo  
06 Teoria-Recursos.txt

## Derivación de clases y herencia - Invalidación de métodos

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

```
Auto a = new Auto();  
Colectivo c = new Colectivo();  
a.Marca = "Ford";  
a.Modelo = 2000;  
c.Marca = "Mercedes";  
c.Modelo = 2010;  
c.CantPasajeros = 20;  
a.Tipo = TipoAuto.Deportivo;  
a.Imprimir();  
c.Imprimir();
```

----- TipoAuto.cs -----

```
enum TipoAuto { Familiar, Deportivo, Camioneta }
```

----- Automotor.cs -----

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

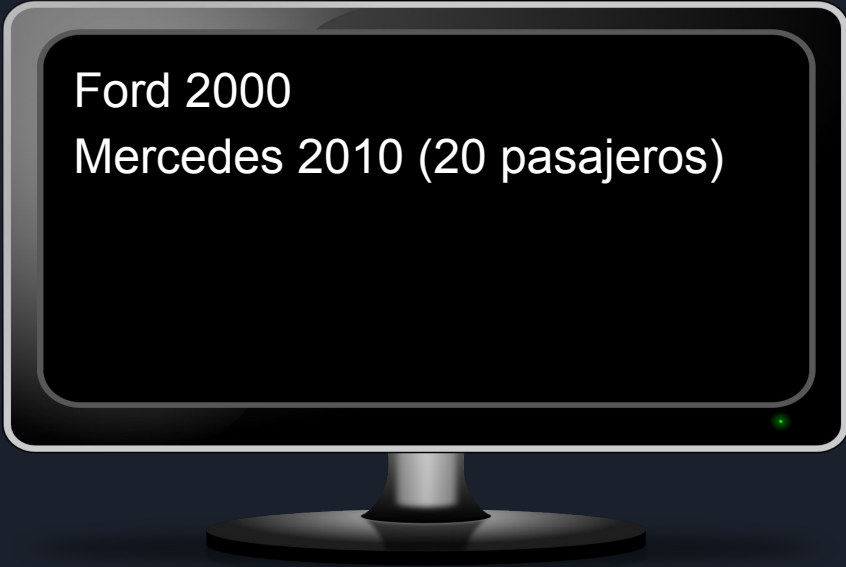
----- Auto.cs -----

```
class Auto : Automotor {  
    public TipoAuto Tipo;  
}
```

----- Colectivo.cs -----

```
class Colectivo : Automotor {  
    public int CantPasajeros;  
    public override void Imprimir()  
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");  
}
```

Colectivo tiene su propio  
método **Imprimir()**  
Auto sigue usando el  
método heredado de su  
clase base



Ford 2000  
Mercedes 2010 (20 pasajeros)



## Acceso a miembros de la clase base

- La palabra clave `base` se utiliza para obtener acceso a los miembros de la `clase base` (la superclase) desde una `clase derivada`. Se utiliza en dos situaciones:
  - Para Invocar a un método (u otro miembro) de la clase base
  - En el encabezado de un constructor para indicar el constructor de la clase base que se debe invocar.





Codificar el método Imprimir() de la clase Auto invocando al método Imrpimir() de Automotor



```
namespace Teoria6;
```

```
class Auto : Automotor
{
    public TipoAuto Tipo;
    public override void Imprimir()
    {
        Console.WriteLine($"Auto {Tipo} ");
        base.Imrpimir();
    }
}
```

Invocación del método  
Imprimir() definido en la  
clase Automotor

Código en el archivo  
06 Teoria-Recursos.txt



## Derivación de clases y herencia - Acceso a miembros de la clase base

----- TipoAuto.cs -----

```
enum TipoAuto { Familiar, Deportivo, Camioneta }
```

----- Automotor.cs -----

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

----- Auto.cs -----

```
class Auto : Automotor {  
    public TipoAuto Tipo;  
    public override void Imprimir()  
    {  
        Console.Write($"Auto {Tipo} ");  
        base.Imprimir();  
    }  
}
```

----- Colectivo.cs -----

```
class Colectivo : Automotor {  
    public int CantPasajeros;  
    public override void Imprimir()  
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");  
}
```

Un clase derivada  
puede aprovechar  
código de la clase  
base aún en el caso  
que esté invalidando el  
método

Auto Deportivo Ford 2000  
Mercedes 2010 (20 pasajeros)

## Constructores

Los constructores no se heredan, sin embargo, debemos estar atentos cuando los definimos en una jerarquía de clases





# Agregar el siguiente constructor a la clase Automotor e intentar compilar



```
class Automotor
{
    public string Marca = "";
    public int Modelo;

    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }

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

# Derivación de clases y herencia - Constructores

----- Automotor.cs -----

```
class Automotor
{
    public string Marca = "";
    public int Modelo;
    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
    public virtual void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo}");
}
```

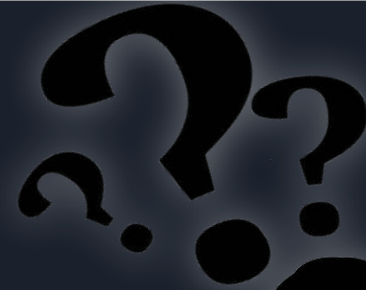
----- Auto.cs -----

```
class Auto : Automotor
{
    public TipoAuto Tipo;
    public override void Imprimir()
    {
        Console.Write($"Auto {Tipo} ");
        base.Imprimir();
    }
}
```

----- Colectivo.cs -----

```
class Colectivo : Automotor
{
    public int CantPasajeros;
    public override void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");
}
```

Error de  
compilación en  
Auto y Colectivo



¿Por qué?

Al definir un constructor en la clase **Automotor**, el compilador ya no coloca el constructor por defecto. Sin embargo sí lo hace con las clases **Auto** y **Colectivo** de la siguiente manera:

```
public Auto() : base() { }
```

y

```
public Colectivo() : base() { }
```

Invoca el constructor sin parámetros de la clase base (que ya no existe)





Probemos agregando el constructor sin argumentos que ya no es incluido automáticamente en la clase Automotor



```
. . .  
public Automotor(string marca, int modelo)  
{  
    Marca = marca;  
    Modelo = modelo;  
}  
public Automotor()  
{  
}  
. . .
```

----- Automotor.cs -----

```
class Automotor
{
    public string Marca = "";
    public int Modelo;
    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
    public Automotor()
    {
    }
    public virtual void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo}");
}
```

----- Auto.cs -----

```
class Auto : Automotor {
    public TipoAuto Tipo;
    public override void Imprimir()
    {
        Console.Write($"Auto {Tipo} ");
        base.Imprimir();
    }
}
```

----- Colectivo.cs -----

```
class Colectivo : Automotor {
    public int CantPasajeros;
    public override void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");
}
```

Solucionado !



----- Automotor.cs -----

```
class Automotor
```

```
{
```

```
    public string Marca = "";
```

```
    public int Modelo;
```

```
    public Auto
```

```
{
```

```
        Marca =
```

```
        Modelo =
```

```
    }
```

```
    public Autom
```

```
{
```

```
}
```

```
    public virtual
```

```
        => Console
```

```
}
```

----- Auto.

```
class Auto : Autom
```

```
    public TipoAuto
```

```
    public override
```

```
{
```

```
        Console.Writ
```

```
        base.Imprimi
```

```
    }
```

```
}
```

```
}
```

----- Colectiv

```
class Colectivo : Auto
```

```
    public int CantPas
```

```
    public override void Imprimir()
```

```
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");
```

```
}
```

En lugar de agregar el constructor sin argumentos en la clase Automotor se pueden definir constructores adecuados en las clases Auto y Colectivo que invoquen al constructor de dos argumentos de la clase Automotor





Agregar el siguiente constructor a la clase Auto



```
. . .  
public Auto(string marca, int modelo, TipoAuto tipo) : base(marca, modelo)  
{  
    this.Tipo = tipo;  
}  
. . .
```



### Agregar el siguiente constructor a la clase Colectivo



```
. . .  
public Colectivo(string marca, int modelo, int cantPasajeros):base(marca, modelo)  
{  
    this.CantPasajeros = cantPasajeros;  
}  
. . .
```



### Modificar Program.cs y ejecutar



```
using Teoria6;
```

```
Auto a = new Auto("Ford", 2000, TipoAuto.Deportivo);
```

```
Colectivo c = new Colectivo("Mercedes", 2010, 20);
```

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

```
c.Imprimir();
```

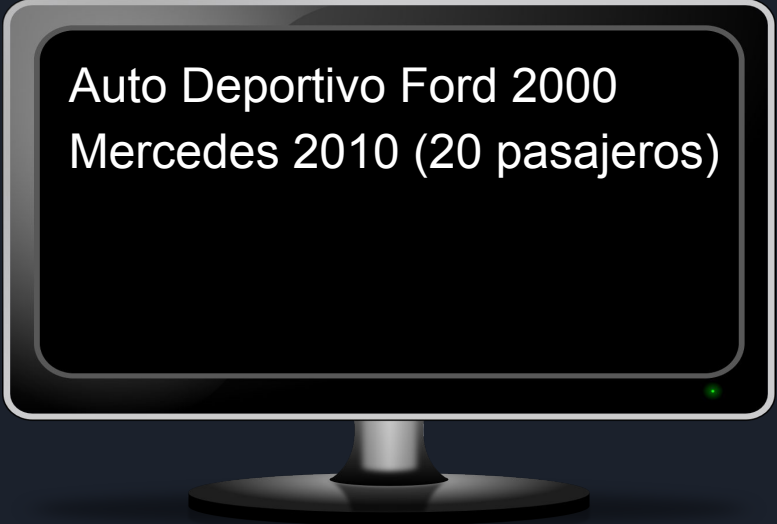
```
using Teoria6;
```

```
Auto a = new Auto("Ford", 2000, TipoAuto.Deportivo);
```

```
Colectivo c = new Colectivo("Mercedes", 2010, 20);
```

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

```
c.Imprimir();
```



Auto Deportivo Ford 2000  
Mercedes 2010 (20 pasajeros)

## Derivación de clases y herencia - Constructores

```
class Automotor (string marca, int modelo)
{
    public string Marca = marca;
    public int Modelo = modelo;

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

Nota: Este sería el código utilizando constructores primarios en las tres clases (sólo a partir de C# 12)

```
class Auto(string marca, int modelo, TipoAuto tipo): Automotor(marca,modelo)
{
    public TipoAuto Tipo=tipo;
    public override void Imprimir()
    {
        Console.Write($"Auto {Tipo} ");
        base.Imprimir();
    }
}
```

Esta es la sintaxis para invocar el constructor de la clase base

```
class Colectivo(string marca, int modelo, int cantPasajeros) : Automotor(marca,modelo)
{
    public int CantPasajeros=cantPasajeros;
    public override void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");
}
```

## Modificadores de acceso

Si bien las clases derivadas heredan todos los miembros de una clase base (a excepción de los constructores y finalizadores), que dichos miembros estén o no visibles depende de su **accesibilidad**.





Establecer los campos Marca y Modelo de la clase Automotor como privados y ejecutar



```
class Automotor
{
    private string Marca;
    private int Modelo;
    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
    public virtual void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo}");
}
```

## Modificadores de acceso

```
class Colectivo : Automotor
{
    public int CantPasajeros;
    public Colectivo(string marca, int modelo, int cantPasajeros)
        : base(marca, modelo) => CantPasajeros = cantPasajeros;

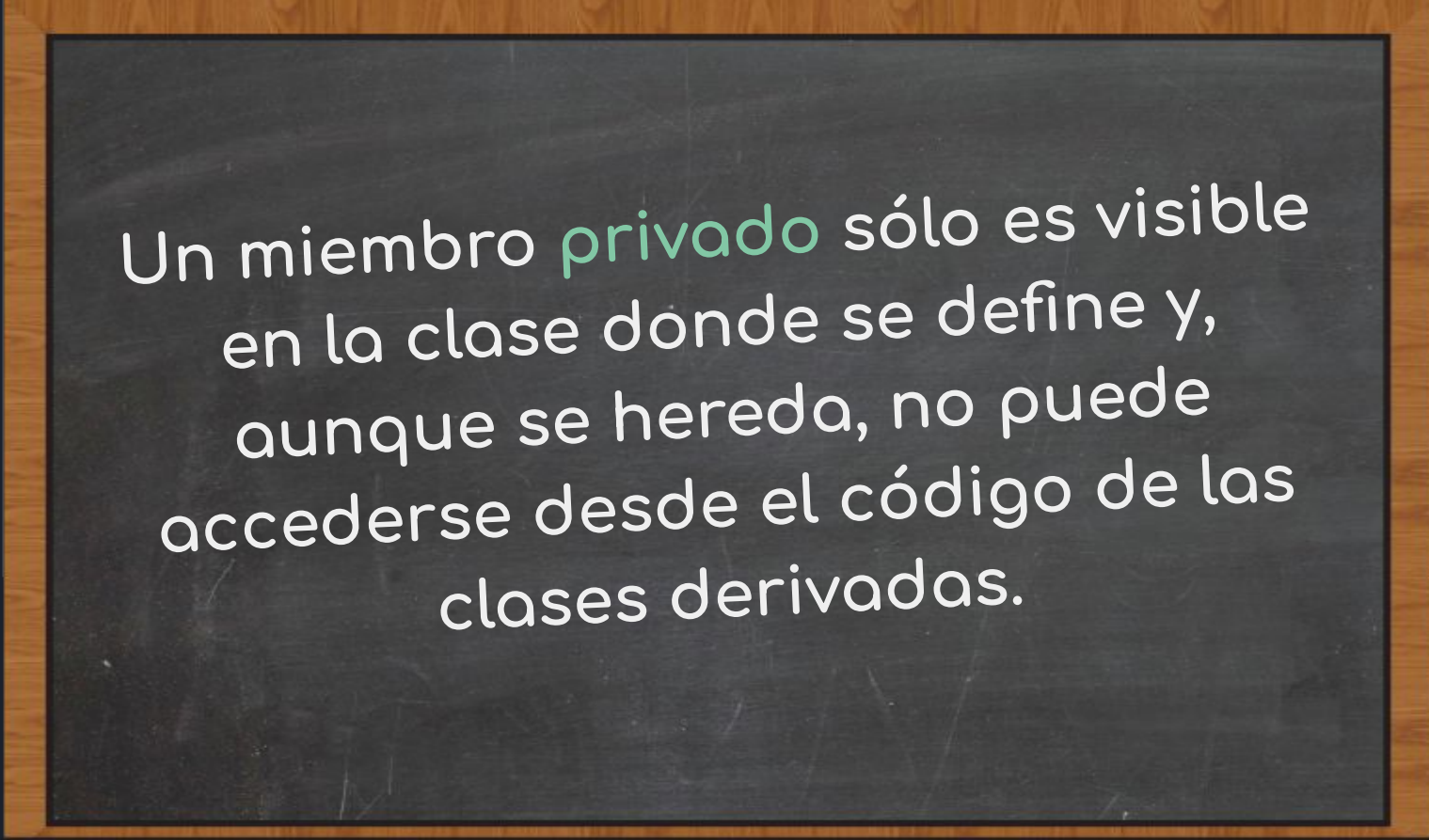
    public override void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");
}
```

Error de  
compilación





## Modificadores de acceso



Un miembro **privado** sólo es visible en la clase donde se define y, aunque se hereda, no puede accederse desde el código de las clases derivadas.

El código de la clase `Auto` no tiene problemas porque nunca accede directamente a los campos definidos en `Automotor`, en su lugar usa invocaciones a la clase base

```
class Auto : Automotor {  
    public Auto(string marca, int modelo, TipoAuto tipo)  
    → : base(marca, modelo) => Tipo = tipo;  
    public override void Imprimir() {  
        Console.WriteLine($"Auto {Tipo} ");  
    → base.Imprimir();  
    ...  
}
```

En general es una buena práctica NO acceder directamente a campos definidos en una superclase

---

Sin embargo, si se necesita extender el acceso de un miembro a las clases derivadas (pero no a las demás), se debe marcar como protegido (**protected**)



## Definir Marca y Modelo como protegidos y ejecutar



```
class Automotor
{
    protected string Marca;
    protected int Modelo;
    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
    public virtual void Imprimir()
        => Console.WriteLine($"{Marca} {Modelo}");
}
```

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

```
Auto a = new Auto("Ford", 2000, TipoAuto.Deportivo);  
Colectivo c = new Colectivo("Mercedes", 2010, 20);  
a.Imprimir(); c.Imprimir();
```

----- TipoAuto.cs -----

```
enum TipoAuto { Familiar, Deportivo, Camioneta }
```

----- Automotor.cs -----

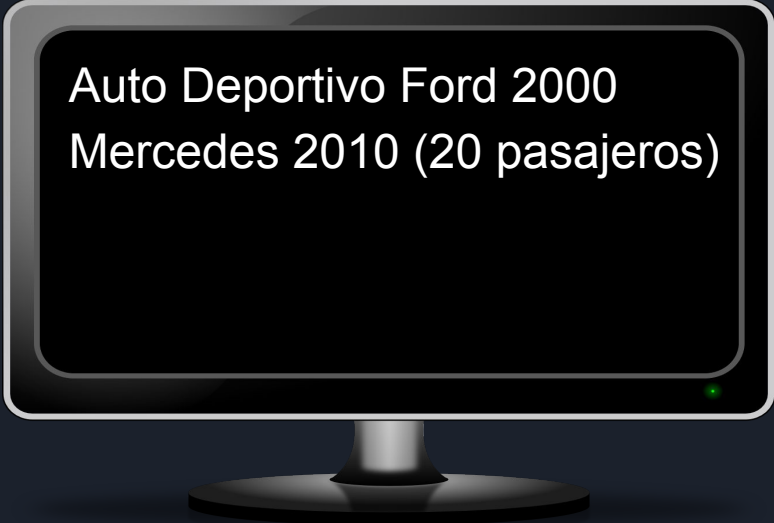
```
class Automotor {  
    protected string Marca = "";  
    protected int Modelo;  
    public Automotor(string marca, int modelo) {  
        Marca = marca;  
        Modelo = modelo;  
    }  
    public virtual void Imprimir() => Console.WriteLine($"{Marca} {Modelo}");  
}
```

----- Auto.cs -----

```
class Auto : Automotor {  
    public TipoAuto Tipo;  
    public Auto(string marca, int modelo, TipoAuto tipo) : base(marca, modelo) => Tipo = tipo;  
    public override void Imprimir() {  
        Console.Write($"Auto {Tipo} ");  
        base.Imprimir();  
    }  
}
```

----- Colectivo.cs -----

```
class Colectivo : Automotor {  
    public int CantPasajeros;  
    public Colectivo(string marca, int modelo, int cantPasajeros)  
        : base(marca, modelo) => CantPasajeros = cantPasajeros;  
    public override void Imprimir()  
        => Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");  
}
```

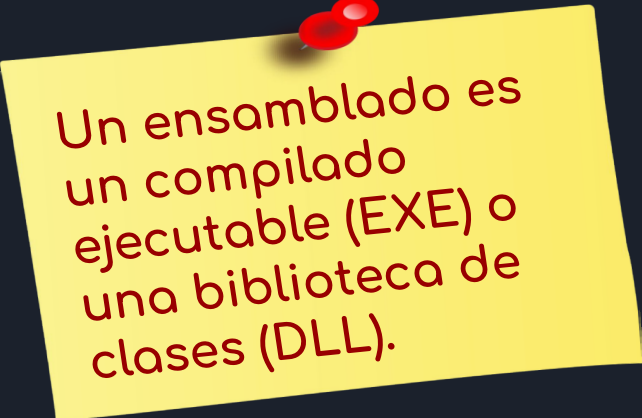


Auto Deportivo Ford 2000  
Mercedes 2010 (20 pasajeros)

Solucionado!

## Modificadores de acceso

- Los miembros de las clases pueden declararse como **públicos**, **internos**, **protegidos** o **privados**.
  - **públicos**: precedidos por el modificador de acceso **public**. Pueden accederse desde cualquier clase de cualquier ensamblado que compone la aplicación.



Un ensamblado es un compilado ejecutable (EXE) o una biblioteca de clases (DLL).



## Modificadores de acceso

- **internos**: precedidos por el modificador de acceso **internal**. pueden accederse sólo desde las clases en el mismo ensamblado (biblioteca de clases o ejecutable).
- **protegidos**: precedidos por el modificador de acceso **protected**. Sólo pueden accederse desde la propia clase o desde sus clases derivadas (estén o no en el mismo ensamblado)
- **privados**: precedidos por el modificador de acceso **private** (por defecto para los miembros de clase). Sólo pueden accederse desde la propia clase.

## Modificadores de acceso

- Las clases pueden ser **públicas** o **internas**
  - **públicas**: precedidas por el modificador de acceso **public**.
  - **internas**: (valor predeterminado para las clases) son precedidas por el modificador de acceso **internal** o no contienen modificador de acceso.
  - No se pueden definir clases con el modificador **private** o **protected** a menos que sea una clase anidada dentro de otra





# Modificar la clase Automotor definiendo las propiedades Marca y Modelo. Ejecutar



```
class Automotor
{
    public string Marca { get; }
    private int _modelo;
    public int Modelo
    {
        get => _modelo;
        set => _modelo = (value < 2005) ? 2005 : value;
    }
    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
    public virtual void Imprimir()
    => Console.WriteLine($"{Marca} {Modelo}");
}
```

Se desea controlar  
que el Modelo sea  
mayor o igual a 2005

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

```
Auto a = new Auto("Ford", 2000, TipoAuto.Deportivo);  
Colectivo c = new Colectivo("Mercedes", 2010, 20);  
a.Imprimir(); c.Imprimir();
```

----- Automotor.cs -----

```
class Automotor {  
    public string Marca { get; }  
    private int _modelo;  
    public int Modelo  
    {  
        get => _modelo;  
        set => _modelo = (value < 2005) ? 2005 : value;  
    }  
    public Automotor(string marca, int modelo)  
    {  
        Marca = marca;  
        Modelo = modelo;  
    }  
    public virtual void Imprimir()  
    => Console.WriteLine($"{Marca} {Modelo}");  
}
```

... .

Inicializada en el constructor

El constructor de Automotor (que también utilizan Auto y Colectivo) utiliza la propiedad Modelo, por lo tanto, si es necesario el modelo será corregido durante la creación del objeto

Auto Deportivo Ford 2005  
Mercedes 2010 (20 pasajeros)



Restringir la escritura de la propiedad Modelo de la clase Automotor.



```
class Automotor
{
    public string Marca { get; }
    private int _modelo;
    public int Modelo
    {
        get => _modelo;
        protected set => _modelo = (value < 2005) ? 2005 : value;
    }
    . . .
}
```

El modificador de acceso afecta sólo al descriptor `set`

La propiedad `Modelo` podrá ser leída por cualquier clase pero sólo podrá modificarse desde la clase `Automotor` y sus derivadas

## Descriptores de acceso con distintos niveles de accesibilidad

```
public int Modelo  
{
```

```
    get => _modelo;
```

```
    protected set => _modelo = (value < 2005)  
                                ? 2005 : value;
```


```
}
```

Se usa un modificador en la propiedad y otro en uno de los *accessors*

El modificador del *accessor* debe ser más restrictivo que el de la propiedad



## Modificadores de acceso (Restricciones)



```
public class Auto : Automotor
{
    public TipoAuto Tipo;
    public Auto(string marca, int modelo, TipoAuto tipo)
        : base(marca, modelo) => Tipo = tipo;

    public override void Imprimir()
    {
        Console.WriteLine($"Auto {Tipo} ");
        base.Imprimir();
    }
}
```

¿Qué pasa si  
declaramos  
pública a la clase  
Auto?

## Modificadores de acceso (Restricciones)

Error de compilación:

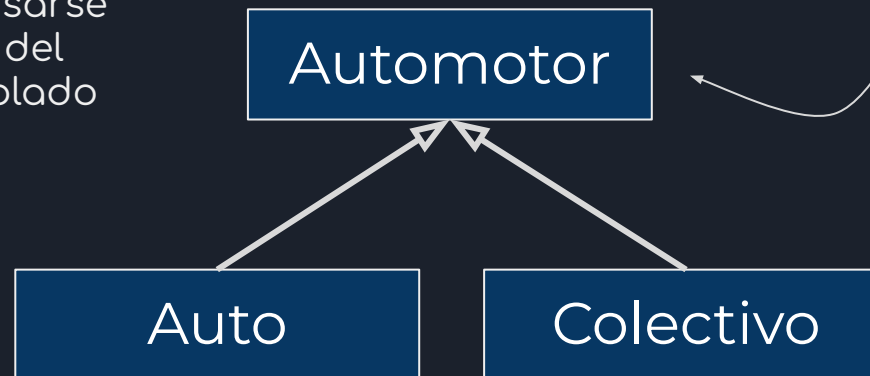
Incoherencia de accesibilidad: la clase base 'Automotor' es menos accesible que la clase 'Auto'

*Pública*

Puede usarse  
fuera del  
ensamblado

*Interna*

Sólo accesible  
en el propio  
ensamblado



## Modificadores de acceso (Restricciones)

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





## Invalidación de propiedades

- Las propiedades, al igual que los métodos, pueden ser redefinidas (invalidadas) en las clases derivadas
- Para ello, al igual que los métodos, la propiedad debe ser marcada con la palabra clave `virtual`
- Esto permite que las clases derivadas invaliden el comportamiento de la propiedad mediante la palabra clave `override`.





# Marcar con virtual a la propiedad Modelo de Automotor (vamos a redefinirla en Colectivo)



```
class Automotor
```

```
{
```

```
    public string Marca { get; }
```

```
    private int _modelo;
```

```
    public virtual int Modelo
```

```
    {
```

```
        get => _modelo;
```

```
        protected set => _modelo = (value < 2005)  
                                     ? 2005 : value;
```

```
    }
```

```
    . . .
```

```
}
```



# Codificar la propiedad Modelo de Colectivo y redefinir el accesor set. Ejecutar



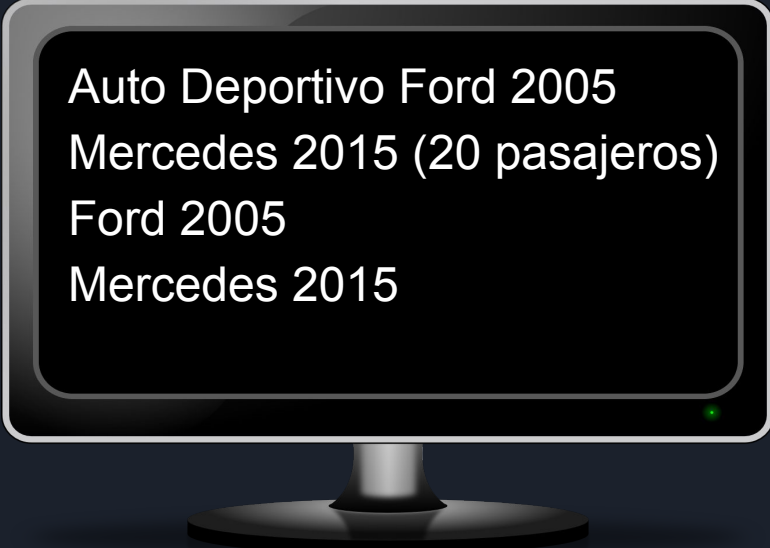
```
class Colectivo : Automotor
{
    public int CantPasajeros;
    public override int Modelo
    {
        protected set =>
            base.Modelo = (value < 2015) ? 2015 : value;
    }
    . . .
}
```

Estamos **invalidando**  
la propiedad Modelo  
de la **clase base**  
(sólo el **accessor set**)

```
----- Program.cs -----  
Auto a = new Auto("Ford", 2000, TipoAuto.Deportivo);  
Colectivo c = new Colectivo("Mercedes", 2010, 20);  
a.Imprimir();  
c.Imprimir();  
Console.WriteLine(a.Marca + " " + a.Modelo);  
Console.WriteLine(c.Marca + " " + c.Modelo);
```

Verificando

Las propiedades  
Marca y Modelo se  
pueden leer de  
manera pública pero  
cualquier intento de  
asignación provocaría  
error de compilación



Auto Deportivo Ford 2005  
Mercedes 2015 (20 pasajeros)  
Ford 2005  
Mercedes 2015

Código en el archivo  
06 Teoria-Recursos.txt

## Clases abstractas

- El propósito de una **clase abstracta** es proporcionar una **definición común de una clase base** que hereden múltiples clases derivadas
- No se pueden crear instancias de una **clase abstracta**
- Las **clases abstractas** se señalan con el modificador **abstract**

## Clases abstractas

- Una **clase abstracta** puede tener **métodos**, **propiedades**, **indizadores** y **eventos** (los veremos más adelante) **abstractos**.
- Una **clase abstracta** también puede tener miembros **no abstractos**.
- Las **miembros abstractos** no tienen implementación, se escriben sin el cuerpo, y llevan el modificador **abstract**.
- Los miembros **abstractos** son también **virtuales** y deben implementarse (**override**) en las subclases concretas.

## Clases abstractas

- El compilador dará error cuando:
  - se declare un **miembro abstracto** en una **clase no abstracta**
  - al **intentar instanciar** un objeto de una clase abstracta
  - si no se implementan los métodos y las propiedades abstractos de la clase base en una clase derivada no abstracta

## Clases abstractas - Ejemplo

```
abstract class Automotor
{
    public abstract void HacerMantenimiento();
    public abstract DateTime FechaService { get; set; }
    public abstract double PrecioDeVenta { get; }
    public string Marca { get; }
    private int _modelo;
    . . .
}
```

Auto y Colectivo deben implementar (override):

- método `HacerMantenimiento()`,
- bloque `get` de la propiedad `PrecioDeVenta`
- bloques `get` y `set` de la propiedad `FechaService`

## Clases abstractas - Ejemplo

¡ ATENCIÓN !

```
public abstract DateTime FechaService {get; set;}  
public abstract double PrecioDeVenta {get;}
```

Son propiedades abstractas, no confundir con propiedades auto-implementadas.

El modificador **abstract** es la clave para interpretar correctamente estas líneas de código





## Derivación y miembros estáticos

- Los **miembros estáticos** también se heredan pero **no pueden invalidarse**.
- Sólo usamos los modificadores **virtual**, **override** o **abstract** con miembros de instancia, de lo contrario obtenemos un error de compilación.
- Más información en la sección “Para conocer más - Contenido opcional” al final de esta teoría.



# Polimorfismo



# Polimorfismo

- El **polimorfismo** suele considerarse el tercer pilar de la programación orientada a objetos, después del **encapsulamiento** y la **herencia**
- Un **objeto polimórfico** en tiempo de **ejecución** puede adoptar la forma de un **tipo no idéntico a su tipo declarado**.
- La relación “**es un**” asociada a la herencia permite tener objetos polimórficos

### Objetos polimórficos - Ejemplo

```
object o;
```

```
o = 1;
```

```
o = "ABC";
```

```
Automotor a;
```

```
a = new Auto("Ford", 2000, TipoAuto.Deportivo);
```

```
a = new Colectivo("Mercedes", 2010, 20);
```

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



## Métodos virtuales y enlace dinámico

- Los **miembros virtuales**, la **invalidación** y el **enlace dinámico** permiten aprovechar el polimorfismo (también las **interfaces** que veremos más adelante).

Sea **a** de tipo **Automotor**. ¿Qué hace la siguiente instrucción?

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



## Métodos virtuales y enlace dinámico

- La respuesta a la pregunta anterior es: “**depende de quien sea a**”
- El método que se invocará en la expresión “**a.Imprimir();**” NO se enlaza estáticamente en tiempo de compilación
- El CLR busca en tiempo de ejecución, el tipo específico del objeto **a**, e invoca la invalidación correcta de **Imprimir()** (**enlace dinámico**)

# Polimorfismo - Consecuencia

El polimorfismo permite que una misma expresión como “`a.Imprimir()`” provoque comportamientos distintos dependiendo de quién sea específicamente `a` en el momento de la invocación.





## Polimorfismo - Consecuencia

El polimorfismo puede ser  
como "a.Imprimir"  
distintos dependiendo de  
a en el momento de ejecución

a.Imprimir()

objeto      mensaje

### REQUISITO

Los objetos que se usan de manera polimórfica deben saber responder al mensaje que se les envía. Con C# el compilador se asegura de ello.

En nuestro ejemplo **a** es un **Automotor** y por lo tanto todo objeto que podamos asignarle conocerá el método **Imprimir()**, la herencia lo garantiza.





# Interfaz polimórfica

- La **interfaz polimórfica** de una **clase base** es el conjunto de sus **miembros virtuales** (recordar que los **abstractos** también son **virtuales**).
- Esto es más interesante de lo que parece a primera vista, ya que permite crear aplicaciones de software fácilmente **extensibles** y **flexibles**.
- Un **diseño adecuado** de la **jerarquía de clases** permitirá tomar **ventaja del polimorfismo**.



## Ejemplo

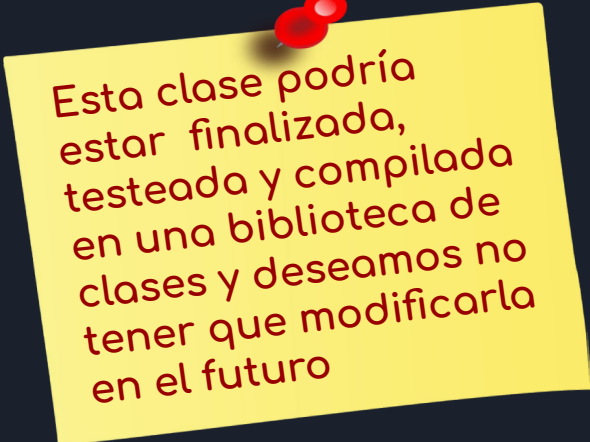
Supongamos que tenemos la clase estática (*utility class*) `ImprimidorAutomotores` dedicada a la impresión de automotores, que usamos de la siguiente manera:

```
Automotor[] vector = [  
    new Auto("Ford", 2015, TipoAuto.Deportivo),  
    new Colectivo("Mercedes", 2019, 20),  
    new Colectivo("Mercedes", 2018, 30),  
    new Auto("Nissan", 2020, TipoAuto.Familiar)  
];  
ImprimidorAutomotores.Imprimir(vector);
```

## Ejemplo

Gracias al **polimorfismo**, el código de la *utility class* es muy sencillo. El programador no necesita tener en cuenta la manera en que se imprime cada subtipo de automotor

```
static class ImprimidorAutomotores
{
    public static void Imprimir(Automotor[] vector)
    {
        foreach (Automotor a in vector)
        {
            a.Imprimir();
        }
    }
}
```



Esta clase podría estar finalizada, testeada y compilada en una biblioteca de clases y deseamos no tener que modificarla en el futuro

## Polimorfismo - Ejemplo

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

```
Automotor[] vector = [  
    new Auto("Ford", 2015, TipoAuto.Deportivo),  
    new Colectivo("Mercedes", 2019, 20),  
    new Colectivo("Mercedes", 2018, 30),  
    new Auto("Nissan", 2020, TipoAuto.Familiar)  
];  
ImprimidorAutomotores.Imprimir(vector);
```

----- ImprimidorAutomotores.cs -----

```
static class ImprimidorAutomotores  
{  
    public static void Imprimir(Automotor[] vector)  
    {  
        foreach (Automotor a in vector)  
        {  
            a.Imprimir();  
        }  
    }  
}
```



Cada automotor se  
imprime de la manera  
correcta, según el  
subtipo específico

### Ejemplo

Supongamos que el sistema está en producción y algún tiempo después se solicita una modificación importante. El sistema ahora debe también trabajar con un nuevo tipo de automotor: las motos.

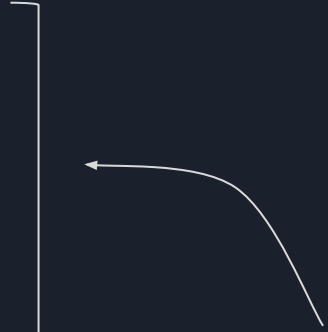
La tarea entonces consiste en agregar la clase Moto derivada de Automotor y sobrescribir (invalidar) convenientemente los miembros necesarios de su interfaz polimórfica



# Ejemplo

```
class Moto : Automotor
{
    public Moto(string marca, int modelo)
        : base(marca, modelo) { }

    public override void Imprimir()
    {
        Console.Write("Moto ");
        base.Imprimir();
    }
    . . .
}
```



Codificamos la manera  
en que una moto se  
imprime a sí misma

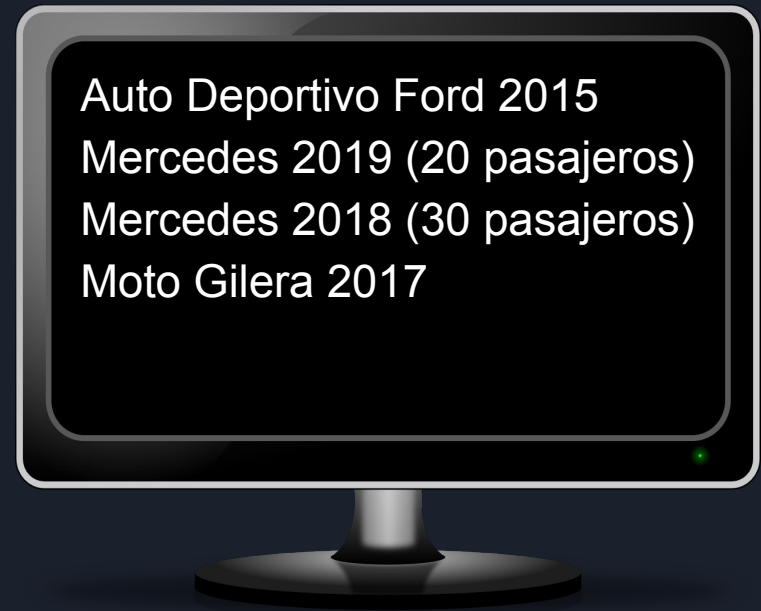
## Polimorfismo - Ejemplo

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

```
Automotor[] vector = [  
    new Auto("Ford", 2015, TipoAuto.Deportivo),  
    new Colectivo("Mercedes", 2019, 20),  
    new Colectivo("Mercedes", 2018, 30),  
    new Moto("Gilera", 2017)  
];  
ImprimidorAutomotores.Imprimir(vector);
```

----- ImprimidorAutomotores.cs -----

```
static class ImprimidorAutomotores  
{  
    public static void Imprimir(Automotor[] vector)  
    {  
        foreach (Automotor a in vector)  
        {  
            a.Imprimir();  
        }  
    }  
}
```



Observar que gracias al  
polimorfismo no tuvimos que  
modificar la clase  
ImprimidorAutomotores

## Principio Open / Close

El principio **Open/Close** (el segundo de los principios **SOLID**) establece:

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

El polimorfismo contribuye a que podamos cumplir con este principio





## Diseño ineficiente que no hace uso de polimorfismo

- Si nuestro código requiere consultar por el tipo de un objeto (operador `is`) puede ser una señal de un diseño ineficiente
- Semántica del operador `is` :
  - Si `a` es una instancia de `Auto`


```
a is Auto // devuelve true
```

```
a is Colectivo //devuelve false
```

```
a is Automotor // devuelve true
```

```
a is object // devuelve true
```

```
a is string // devuelve false
```



## Diseño ineficiente que no hace uso de polimorfismo

Caso hipotético de un diseño ineficiente:

- No definimos la clase `Automotor` con su interfaz polimórfica.
- `Auto` y `Colectivo` derivan directamente de `object`.
- Los métodos `Imprimir()` de `Auto` y `Colectivo` no están relacionados entre sí, incluso sus nombres podrían no coincidir.
- `Imprimir()` de la clase `ImprimidorAutomotores` no puede tomar ventaja del polimorfismo

## Diseño ineficiente que no hace uso de polimorfismo

```
static class ImprimidorAutomotores {  
    public static void Imprimir(object[] vector) {  
        foreach (object o in vector)  
        {  
            if (o is Auto)  
            {  
                (o as Auto).Imprimir();  
            }  
            else if (o is Colectivo)  
            {  
                (o as Colectivo).Imprimir();  
            }  
        }  
    }  
}
```

Es necesario consultar  
por el tipo de los objetos  
para poder convertirlos  
adecuadamente

## Diseño ineficiente que no hace uso de polimorfismo

```
static class ImprimidorAutomotores {  
    public static void Imprimir(object[] vector) {  
        foreach (object o in vector)  
        {  
            if (o is Auto)  
            {  
                (o as Auto).Imprimir();  
            }  
            else if (o is Colectivo)  
            {  
                (o as Colectivo).Imprimir();  
            }  
        }  
    }  
}
```

Esta clase no está cerrada para la modificación.  
Al agregar la clase Moto, también se debería modificar este método

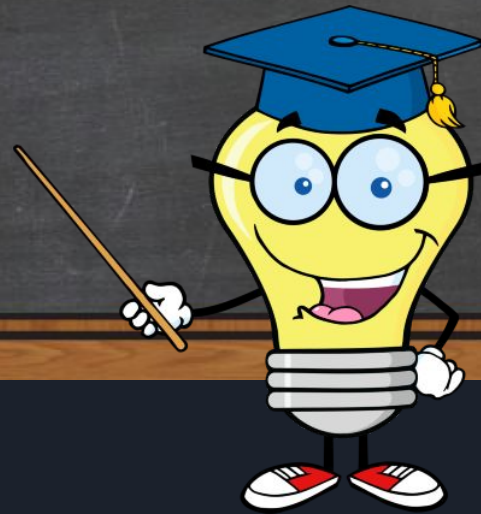
# Nota

A partir de la versión 7.0 de C# el código

```
if (o is Auto)
{
    (o as Auto).Imprimir();
}
```

Se puede simplificar de la siguiente manera:

```
if (o is Auto a)
{
    a.Imprimir();
}
```



## Ejemplo de buen uso del polimorfismo

### El método ToString()

- Uno de los métodos de la interfaz polimórfica de `object` es:

```
public virtual string ToString();
```

- La implementación genérica en `object` simplemente devuelve el nombre del tipo del objeto que recibe el mensaje, es lo que muestra el método `WriteLine()` de la clase `Console`.

## Ejemplo de buen uso del polimorfismo

### El método ToString()

- Por ejemplo si `a` es una instancia de `Auto`, la instrucción `Console.WriteLine(a)` imprime en la consola `Teoria6.Auto`
- Gracias al `polimorfismo`, no es necesario modificar el método `WriteLine()` de la clase `Console` para cambiar este comportamiento. Podemos hacerlo invalidando el método `ToString()` de la clase `Auto` (principio `Open/Close`)

## Polimorfismo - El método ToString() - Ejemplo

```
----- Program.cs -----  
Auto a = new Auto("Fiat", 2017, TipoAuto.Familiar);  
Console.WriteLine(a);
```

```
----- Automotor.cs -----  
abstract class Automotor  
{  
    public override string ToString()  
    {  
        return $"Marca: {Marca} \nModelo: {Modelo}";  
    }  
    ...  
}
```

```
----- Auto.cs -----  
class Auto : Automotor  
{  
    public override string ToString()  
    {  
        return base.ToString() + $" \nTipo: {Tipo}";  
    }  
    ...  
}
```





# Notas complementarias



## Excepciones definidas por el usuario

- .NET proporciona una **jerarquía de clases de excepciones** derivadas en última instancia de la clase base **Exception**.
- También podemos crear nuestras propias excepciones **derivando clases de Exception**. La convención es terminar el nombre de clase con la palabra "Exception" e implementar los tres constructores comunes

## Excepciones definidas por el usuario

Ejemplo:

```
public class TanqueCapacidadExcedidaException : Exception {  
    public TanqueCapacidadExcedidaException() { }
```

```
    public TanqueCapacidadExcedidaException(string message)  
        : base(message) { }
```

```
    public TanqueCapacidadExcedidaException(string message,  
        Exception inner): base(message, inner) { }
```

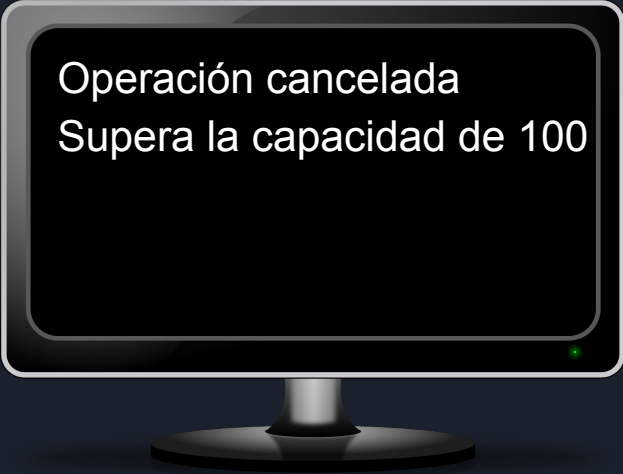
```
    . . .
```

```
}
```

Se usa cuando en un manejador de excepción (bloque catch) se lanza otra excepción pero en inner se mantiene la referencia a la original

```
----- Program.cs -----  
Tanque t=new Tanque();  
try {  
    t.Agregar(200);  
} catch(TanqueCapacidadExcedidaException e) {  
    Console.WriteLine($"Operación cancelada\n{e.Message}");  
}
```

```
----- Tanque.cs -----  
class Tanque {  
    public double Capacidad { get; set; } = 100;  
    public double Litros { private set; get; }  
    public void Agregar(double litros) {  
        if ((this.Litros + litros) > Capacidad) {  
            string mensaje = $"Supera la capacidad de {Capacidad}";  
            throw new TanqueCapacidadExcedidaException(mensaje);  
        } else this.Litros += litros;  
    }  
}
```



Operación cancelada  
Supera la capacidad de 100



## Extensión de métodos

- La **herencia** permite extender el funcionamiento de clases existentes, sin embargo no siempre está disponible
- No pueden derivarse **clases selladas** (marcadas con el modificador **sealed**) ni tampoco estructuras (por ejemplo los tipos numéricos)
- Los **métodos de extensión** permiten agregar funcionalidad a los tipos incluso en los casos en que la herencia no está permitida



## Extensión de métodos

- Los **métodos de extensión** son métodos con los que se extiende a un tipo pero se definen realmente como miembros en otro tipo (una clase estática).
- Supongamos que queremos contar con un **conjunto extra de funciones** para trabajar con tipos **int**.
- Una buena idea es definir una clase estática que agrupe las nuevas funciones.

## Extensión de métodos

Definimos la siguiente *utility class* con tres métodos que nos interesan

```
static class IntExtension
{
    public static int Factorial(int n)
        => (n == 0) ? 1 : n * Factorial(n - 1);
    public static bool EsDivisible(int a, int b)
        => (a % b == 0);
    public static bool SeEscribeComo(int a, string st)
        => (a.ToString() == st);
}
```

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

```
Console.WriteLine(IntExtension.Factorial(5));  
Console.WriteLine(IntExtension.EsDivisible(10, 5));  
Console.WriteLine(IntExtension.EsDivisible(10, 3));  
Console.WriteLine(IntExtension.SeEscribeComo(10, "10"));
```

----- IntExtension.cs -----

```
static class IntExtension  
{  
    public static int Factorial(int n)  
        => (n == 0) ? 1 : n * Factorial(n - 1);  
    public static bool EsDivisible(int a, int b)  
        => (a % b == 0);  
    public static bool SeEscribeComo(int a, string st)  
        => (a.ToString() == st);  
}
```

Por ahora sólo  
es una clase de  
utilidad



120  
True  
False  
True

Queremos convertir estos métodos en  
métodos de extensión para usarlos con  
la sintaxis de métodos de instancia



## Extensión de métodos

El primer parámetro especifica en qué tipo funciona el método.

```
static class IntExtension  
{
```

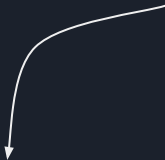
```
    public static int Factorial(this int n)  
        => (n == 0) ? 1 : n * Factorial(n - 1);
```

```
    public static bool EsDivisible(this int a, int b)  
        => (a % b == 0);
```

```
    public static bool SeEscribeComo(this int a, string st)  
        => (a.ToString() == st);
```

```
}
```

anteponer el  
modificar this al  
primer parámetro



## Notas complementarias - Extensión de métodos

```
----- Program.cs -----  
Console.WriteLine(5.Factorial());  
Console.WriteLine(10.EsDivisible(5));  
Console.WriteLine(10.EsDivisible(3));  
Console.WriteLine(10.SeEscribeComo("10"));
```

Sintaxis  
invocación  
métodos de  
instancia

```
----- IntExtension.cs -----  
static class IntExtension  
{  
    public static int Factorial(this int n)  
        => (n == 0) ? 1 : n * Factorial(n - 1);  
    public static bool EsDivisible(this int a, int b)  
        => (a % b == 0);  
    public static bool SeEscribeComo(this int a, string st)  
        => (a.ToString() == st);  
}
```

El nombre de la  
clase ahora es  
irrelevante



120  
True  
False  
True

## Extensión de métodos

Si la clase estática con los métodos de extensión se encuentra definida en otro espacio de nombres, estos métodos estarán disponibles únicamente cuando el espacio de nombre se importe explícitamente con la directiva `using`



## Extensión de métodos

Algunos programadores al codificar un método de extensión nombran al primer parámetro con el identificador `@this`, para hacer más evidente que se trata de una extensión de método. Por ejemplo:

```
static class IntExtension
{
    public static int Factorial(this int @this)
        => (@this == 0) ? 1 : @this * Factorial(@this - 1);
    public static bool EsDivisible(this int @this, int b)
        => (@this % b == 0);
    public static bool SeEscribeComo(this int @this, string st)
        => (@this.ToString() == st);
}
```



## Nota sobre clases estáticas y métodos de extensión

- Las **clases estáticas** no se pueden derivar (son implícitamente selladas) ni pueden heredar de ninguna clase (excepto **System.Object**)
- Los **métodos de extensión** solo extienden métodos de instancia. No se pueden extender métodos estáticos



# Fin Teoría 6





- **Herencia para Relaciones "Es Un" (Is-A):**
  - Utilizar la herencia sólo cuando una clase derivada es *verdaderamente un tipo especializado* de la clase base (un `Auto` es un `Automotor`).
  - Evitar usar herencia para relaciones "Tiene Un" (Has-A) o para simple reutilización de código; la composición suele ser más flexible en esos casos
- **Usar**
  - Declarar una clase `abstract` si no tiene sentido crear instancias directas de ella, pero sirve como base común (ej: `Automotor` si no existen "automotores genéricos").
  - Declarar métodos o propiedades `abstract` es una forma de definir un contrato obligatorio para todas las clases derivadas concretas.
- **Controlar la Visibilidad**
  - Mantener los campos `private` en la clase base.
  - Exponer la funcionalidad necesaria a las clases derivadas a través de métodos o propiedades `public` o `protected`.

- **Declarar `virtual` con Intención:**

- Marca un método o propiedad como `virtual` en la clase base *solo si se espera* que las clases derivadas puedan cambiar o extender su comportamiento. No hacer todo `virtual` por defecto.

- **Aprovechar el Polimorfismo para Evitar Chequeos de Tipo:**

- El gran poder del polimorfismo (usando `virtual/override` o `abstract/override`) es tratar objetos de clases derivadas a través de una referencia de la clase base, invocando la implementación correcta en tiempo de ejecución.
- Esto evita código frágil lleno de `if (obj is Auto) ...`

- **Reutilizar Código Base**

- Dentro de un método `override`, si se necesita ejecutar la lógica original de la clase base, invocarla explícitamente usando `base.NombreMetodo()`.

- **Sobrescribir**

- Sobrescribir (`override`) el método `ToString()` en nuestras propias clases para proporcionar una representación textual significativa del estado del objeto. Es invaluable para debugging y logging.

- **Definir Excepciones Personalizadas (si aplica):**

- Si nuestra jerarquía de clases maneja situaciones de error específicas, considerar crear excepciones personalizadas para mejorar la claridad y el manejo de errores.





# Práctica sobre la teoría 6

1) Sin borrar ni modificar ninguna línea, completar la definición de las clases **B**, **C** y **D**

```
class A
{
    protected int _id;
    public A(int id) => _id = id;
    public virtual void Imprimir() => Console.WriteLine($"A_{_id}");
}

class B : A
{
    . . .
}

class C : B
{
    . . .
}

class D : C
{
    . . .
    public override void Imprimir()
    {
        . . .
        base.Imprimir();
    }
}
```

Para que el siguiente código produzca la salida indicada:

```
A[] vector = [new A(3),new B(5),new C(15),new D(41)];  
foreach (A a in vector)  
{  
    a.Imprimir();  
}
```

Salida por consola

```
A_3  
B_5 --> A_5  
C_15 --> B_15 --> A_15  
D_41 --> C_41 --> B_41 --> A_41
```

2) Aunque consultar en el código por el tipo de un objeto indica habitualmente un diseño ineficiente, por motivos didácticos vamos a utilizarlo. Completar el siguiente código, que utiliza las clases definidas en el ejercicio anterior, para que se produzca la salida indicada:

```
A[] vector = [ new C(1),new D(2),new B(3),new D(4),new B(5)];  
foreach (A a in vector)  
{  
    ...  
}
```

Salida por consola

```
B_3 --> A_3  
B_5 --> A_5
```

Es decir, se deben imprimir sólo los objetos cuyo tipo exacto sea **B**

- Utilizando el operador **is**
- Utilizando el método **GetType()** y el operador **typeof()** (investigar sobre éste último en la documentación en línea de .net)

3) ¿Por qué no funciona el siguiente código? ¿Cómo se puede solucionar fácilmente?

```
class Auto
{
    double velocidad;
    public virtual void Acelerar() => Console.WriteLine("Velocidad = {0}", velocidad += 10);
}

class Taxi : Auto
{
    public override void Acelerar() => Console.WriteLine("Velocidad = {0}", velocidad += 5);
}
```

4) Contestar sobre el siguiente programa:

```
Taxi t = new Taxi(3);
Console.WriteLine($"Un {t.Marca} con {t.Pasajeros} pasajeros");

class Auto
{
    public string Marca { get; private set; } = "Ford";
    public Auto(string marca) => this.Marca = marca;
    public Auto() { }
}

class Taxi : Auto
{
    public int Pasajeros { get; private set; }
    public Taxi(int pasajeros) => this.Pasajeros = pasajeros;
}
```

¿Por qué no es necesario agregar **:base** en el constructor de **Taxi**? Eliminar el segundo constructor de la clase **Auto** y modificar la clase **Taxi** para el programa siga funcionando.

5) ¿Qué líneas del siguiente código provocan error de compilación y por qué?

```
class Persona
{
    public string Nombre { get; set; }
}
public class Auto
{
    private Persona _dueño1, _dueño2;
    public Persona GetPrimerDueño() => _dueño1;
    protected Persona SegundoDueño
    {
        set => _dueño2 = value;
    }
}
```

6) Señalar el error en cada uno de los siguientes casos:

(6.1)

```
class A
{
    public string M1() => "A.M1";
}
class B : A
{
    public override string M1() => "B.M1";
}
```

(6.2)

```
class A
{
    public abstract string M1();
}
class B : A
{
    public override string M1() => "B.M1";
}
```

## Práctica sobre la teoría 6

(6.3)

```
abstract class A
{
    public abstract string M1() => "A.M1";
}
class B : A
{
    public override string M1() => "B.M1";
}
```

(6.4)

```
class A
{
    public override string M1() => "A.M1";
}
class B : A
{
    public override string M1() => "B.M1";
}
```

(6.5)

```
class A
{
    public virtual string M1() => "A.M1";
}
class B : A
{
    protected override string M1() => "B.M1";
}
```

(6.6)

```
class A
{
    public static virtual string M1() => "A.M1";
}
class B : A
{
    public static override string M1() => "B.M1";
}
```

(6.7)

```
class A
{
    virtual string M1() => "A.M1";
}
class B : A
{
    override string M1() => "B.M1";
}
```

(6.8)

```
class A
{
    protected A(int i) { }
}
class B : A
{
    B() { }
}
```

(6.9)

```
class A
{
    private int _id;
    protected A(int i) => _id = i;
}
class B : A
{
    B(int i):base(5) {
        _id = i;
    }
}
```

(6.10)

```
class A
{
    private int Propiedad { set; public get; }
}

class B : A
{
}
```

(6.11)

```
abstract class A
{
    public abstract int Prop {set; get;}
}
class B : A
{
    public override int Prop
    {
        get => 5;
    }
}
```

(6.12)

```
abstract class A
{
    public int Prop {set; get;}
}
class B : A
{
    public override int Prop {
        get => 5;
        set {}
    }
}
```

## Práctica sobre la teoría 6

7) Ofrecer una implementación polimórfica para mejorar el siguiente programa:

```
Imprimidor.Imprimir(new A(), new B(), new C(), new D());

class A {
    public void ImprimirA() => Console.WriteLine("Soy una instancia A");
}

class B {
    public void ImprimirB() => Console.WriteLine("Soy una instancia B");
}

class C {
    public void ImprimirC() => Console.WriteLine("Soy una instancia C");
}

class D {
    public void ImprimirD() => Console.WriteLine("Soy una instancia D");
}

static class Imprimidor
{
    public static void Imprimir(params object[] vector)
    {
        foreach (object o in vector)
        {
            if (o is A a) { a.ImprimirA(); }
            else if (o is B b) { b.ImprimirB(); }
            else if (o is C c) { c.ImprimirC(); }
            else if (o is D d) { d.ImprimirD(); }
        }
    }
}
```



## Práctica sobre la teoría 6

8) Crear un programa para gestionar empleados en una empresa. Los empleados deben tener las propiedades públicas de sólo lectura **Nombre**, **DNI**, **FechaDeIngreso**, **SalarioBase** y **Salario**. Los valores de estas propiedades (a excepción de **Salario** que es una propiedad calculada) deben establecerse por medio de un constructor adecuado.

Existen dos tipos de empleados: **Administrativo** y **Vendedor**. No se podrán crear objetos de la clase padre **Empleado**, pero sí de sus clases hijas (**Administrativo** y **Vendedor**). Aparte de las propiedades de solo lectura mencionadas, el administrativo tiene otra propiedad pública de lectura/escritura llamada **Premio** y el vendedor tiene otra propiedad pública de lectura/escritura llamada **Comision**.

La propiedad de solo lectura **Salario**, se calcula como el salario base más la comisión o el premio según corresponda.

Las clases tendrán además un método público llamado **AumentarSalario()** que tendrá una implementación distinta en cada clase. En el caso del administrativo se incrementará el salario base en un 1% por cada año de antigüedad que posea en la empresa, en el caso del vendedor se incrementará el salario base en un 5% si su antigüedad es inferior a 10 años o en un 10% en caso contrario.

El siguiente código (ejecutado el día 9/4/2022) debería mostrar en la consola el resultado indicado:

```
Empleado[] empleados = new Empleado[] {  
    new Administrativo("Ana", 20000000, DateTime.Parse("26/4/2018"), 10000) {Premio=1000},  
    new Vendedor("Diego", 30000000, DateTime.Parse("2/4/2010"), 10000) {Comision=2000},  
    new Vendedor("Luis", 33333333, DateTime.Parse("30/12/2011"), 10000) {Comision=2000}  
};  
foreach (Empleado e in empleados)  
{  
    Console.WriteLine(e);  
    e.AumentarSalario();  
    Console.WriteLine(e);  
}
```

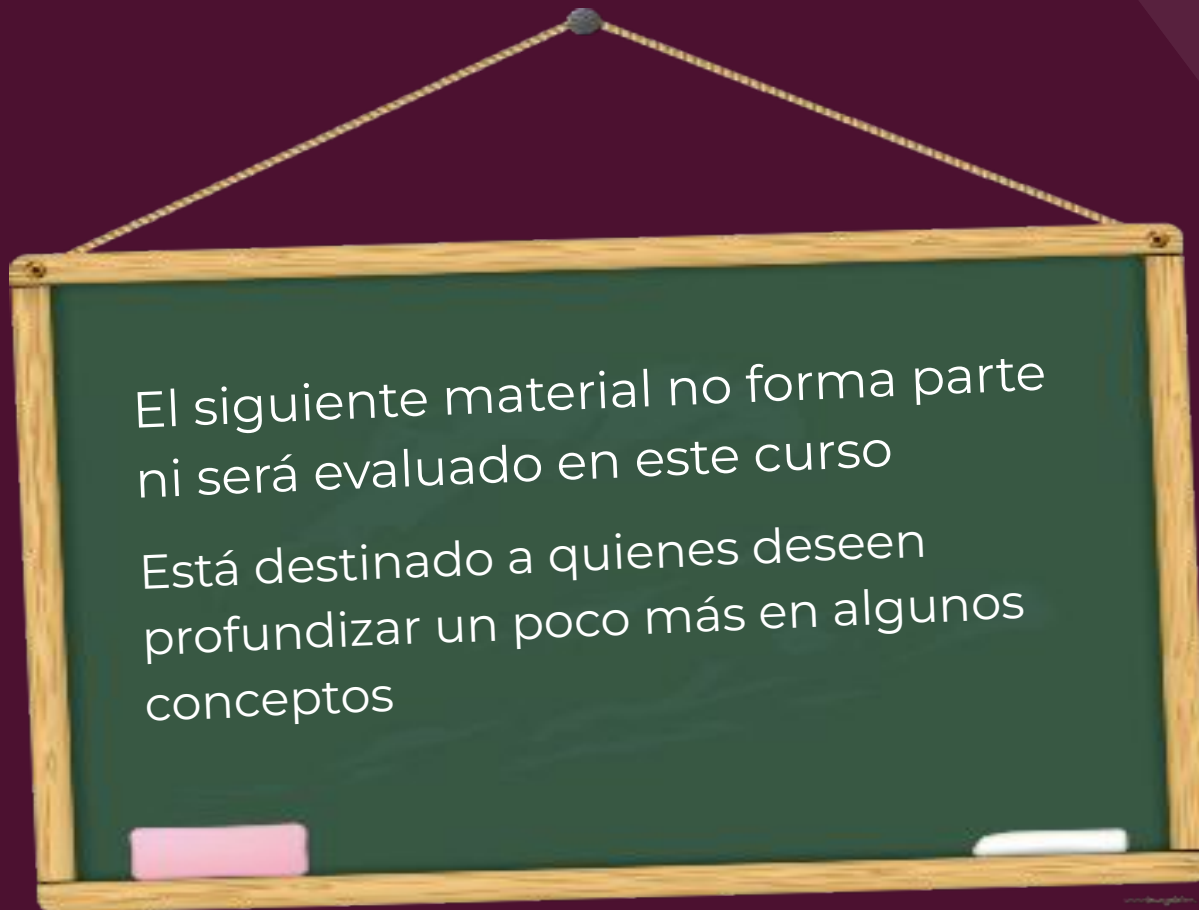
### Salida por consola

```
Administrativo Nombre: Ana, DNI: 20000000 Antigüedad: 3
Salario base: 10000, Salario: 11000
-----
Administrativo Nombre: Ana, DNI: 20000000 Antigüedad: 3
Salario base: 10300, Salario: 11300
-----
Vendedor Nombre: Diego, DNI: 30000000 Antigüedad: 12
Salario base: 10000, Salario: 12000
-----
Vendedor Nombre: Diego, DNI: 30000000 Antigüedad: 12
Salario base: 11000, Salario: 13000
-----
Vendedor Nombre: Luis, DNI: 33333333 Antigüedad: 10
Salario base: 10000, Salario: 12000
-----
Vendedor Nombre: Luis, DNI: 33333333 Antigüedad: 10
Salario base: 11000, Salario: 13000
-----
```

**Recomendaciones:** Observar que el método **AumentarSalario()** y la propiedad de solo lectura **Salario** en la clase **Empleado** pueden declararse como abstractos. Intentar no usar campos sino propiedades auto-implementadas todas las veces que sea posible. Además sería deseable que la propiedad **SalarioBase** definida en **Empleado** sea pública para la lectura y protegida para la escritura, para que pueda establecerse desde las subclases **Administrativo** y **Vendedor**.

# Para conocer más

## Contenido opcional



# Modificadores de acceso combinados

- Además existen las siguientes combinaciones de modificadores de acceso:
  - `protected internal`: Pueden accederse desde cualquier código del ensamblado, o desde una clase derivada de cualquier ensamblado
  - `private protected`: Sólo pueden accederse desde la propia clase o desde sus clases derivadas en el mismo ensamblado

## Finalizadores (destructores)

- Los **finalizadores** (también conocidos como **destructores**) se usan para “hacer limpieza” cuando el recolector de elementos no utilizados (*garbage collector*) libera memoria.
- El *garbage collector* comprueba periódicamente si hay objetos no referenciados por ninguna aplicación. En tal caso llama al finalizador (si existe) y libera la memoria que ocupaba el objeto.
- Se puede forzar la recolección con **GC.Collect()** pero en general debe evitarse por razones de rendimiento.

## Finalizadores (destructores)

- En las aplicaciones de **.NET Framework** (pero no en las de **.NET Core**), cuando se cierra el programa también se llama a los finalizadores.
- No se puede llamar a los finalizadores, **se invocan automáticamente**.
- Puede haber **un solo** finalizador **por clase**, no puede tener **ni parámetros ni modificadores**.
- Los finalizadores no se heredan

## Finalizadores (destructores)

- Sintaxis:

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

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

## Para conocer más - Finalizadores - Ejemplo

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

```
for (int i = 0; i < 20; i++)  
{  
    new ClaseC();  
}
```

Se crean y pierden  
objetos *ClaseC*

----- ClaseA.cs -----

```
class ClaseA {  
    static int s_id=1;  
    protected int Id;  
    public ClaseA() => Id = s_id++;  
    ~ClaseA() => Console.WriteLine($"Fin A {Id}");  
}
```

----- ClaseB.cs -----

```
class ClaseB : ClaseA {  
    ~ClaseB() => Console.WriteLine($"Fin B {Id}");  
}
```

----- ClaseC.cs -----

```
class ClaseC : ClaseB {  
    int[] vector = new int[100_000];  
    ~ClaseC() {  
        Console.WriteLine("-----");  
        Console.WriteLine($"Fin C {Id}");  
    }  
}
```

Sólo para ocupar  
espacio de memoria

-----  
Fin C 3  
Fin B 3  
Fin A 3  
-----  
Fin C 2  
Fin B 2  
Fin A 2  
-----  
Fin C 1  
Fin B 1  
Fin A 1  
-----  
Fin C 8  
Fin B 8  
Fin A 8

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



## Finalizadores (destructores)

- El **Garbage Collector (GC)** gestiona la memoria de los objetos **administrados**.
- Algunos objetos, sin embargo, manejan **recursos no administrados** directamente (ej: identificadores de archivos del Sistema Operativo, conexiones de red crudas, etc.). El **GC no** sabe cómo liberar estos recursos.
- Los **finalizadores** (`~NombreClase()`) son un mecanismo especial que el **GC** puede llamar de forma **no determinista** (en un momento impredecible) **después** de que un objeto deja de ser referenciado.
- Su propósito principal es actuar como una **salvaguarda** o **último recurso** para intentar liberar esos recursos **no administrados** si no se hizo de otra forma.



## Finalizadores (destructores)

- **Importante:** El uso directo de finalizadores es **poco común** en C# moderno. Tienen implicaciones de rendimiento y su ejecución no está garantizada en todos los escenarios (especialmente en .NET Core/.NET 5+ al cerrar la app).
- **Nota:** El mecanismo **principal, recomendado y determinista** para gestionar recursos (tanto administrados que encapsulan recursos no administrados - por ejemplo `FileStream` - como no administrados directamente) es la interfaz `IDisposable` y la sentencia `using`, que se verán en detalle en la clase sobre **Interfaces**.





## Herencia de campos estáticos

- Los campos estáticos también se heredan
- Sin embargo, a diferencia de los campos de instancia, la clase derivada no obtiene su propio campo sino que accede al mismo campo que su clase base

## Para conocer más - Herencia de campos estáticos

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

```
Console.WriteLine($"{Animal.Info} - {Perro.Info}");  
Perro.Info = "Perro";  
Console.WriteLine($"{Animal.Info} - {Perro.Info}");
```

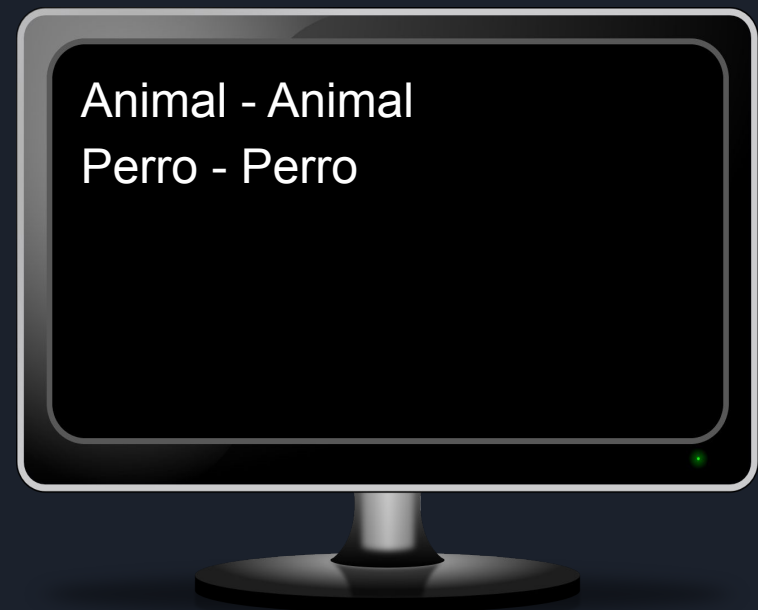
----- Animal.cs -----

```
class Animal  
{  
    public static string Info = "Animal";  
}
```

----- Perro.cs -----

```
class Perro : Animal  
{  
}
```

*Animal.Info y  
Perro.Info  
son la misma  
variable*



## Herencia de campos estáticos

- Es posible volver a definir un campo estático en una clase derivada.
- De esta forma se dice que el nuevo campo oculta al campo de la clase base
- Se debe utilizar el modificador `new` para evitar un `Warning` del compilador

## Para conocer más - Herencia de campos estáticos

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

```
Console.WriteLine($"{Animal.Info} - {Perro.Info}");  
Perro.Info = "PERRO";  
Console.WriteLine($"{Animal.Info} - {Perro.Info}");
```

----- Animal.cs -----

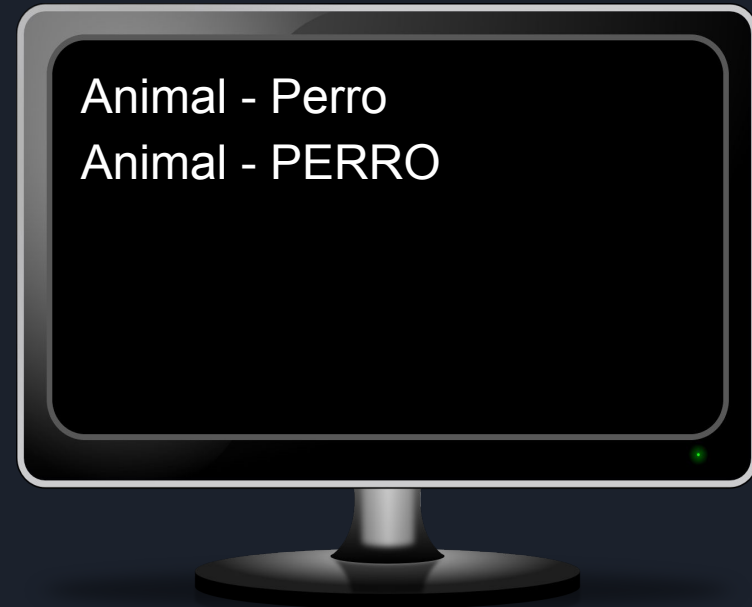
```
class Animal {  
    public static string Info = "Animal";  
}
```

----- Perro.cs -----

```
class Perro : Animal {  
    new public static string Info = "Perro";  
}
```

`new` evita el `warning` del compilador que nos advierte que estamos ocultando el campo heredado `Animal.Info`

`Animal.Info` y  
`Perro.Info`  
son distintas  
variables



## Herencia de métodos estáticos

- Los **métodos estáticos** también se heredan pero **no pueden invalidarse**
- Marcar un método estático como **virtual**, **override** o **abstract** provoca error de compilación
- Los métodos estáticos **pueden ser ocultos** en las clases derivadas

## Para conocer más - Herencia de métodos estáticos

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

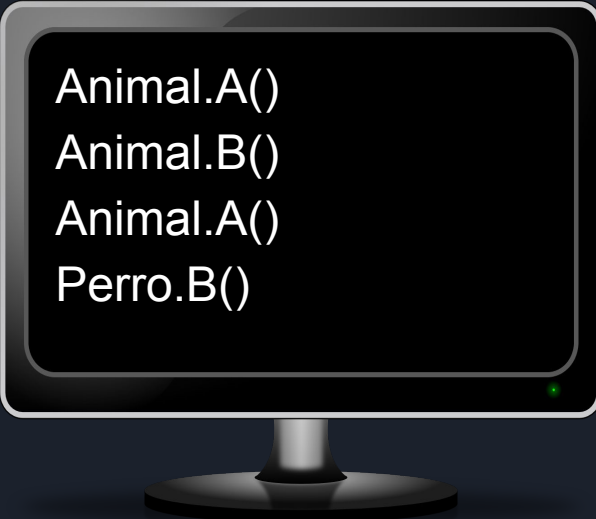
```
Animal.A();  
Animal.B();  
Perro.A();  
Perro.B();
```

----- Animal.cs -----

```
class Animal {  
    public static string Info = "Animal";  
    public static void A() => Console.WriteLine("Animal.A()");  
    public static void B() => Console.WriteLine("Animal.B()");  
}
```

----- Perro.cs -----

```
class Perro : Animal {  
    new public static string Info = "Perro";  
    new public static void B() =>  
        Console.WriteLine("Perro.B()");  
}
```



```
Animal.A()  
Animal.B()  
Animal.A()  
Perro.B()
```

`new` evita el `warning` del compilador que nos advierte que estamos ocultando el método heredado `Animal.B()`