

C#

Clases - Continuación

Derivación de clases

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

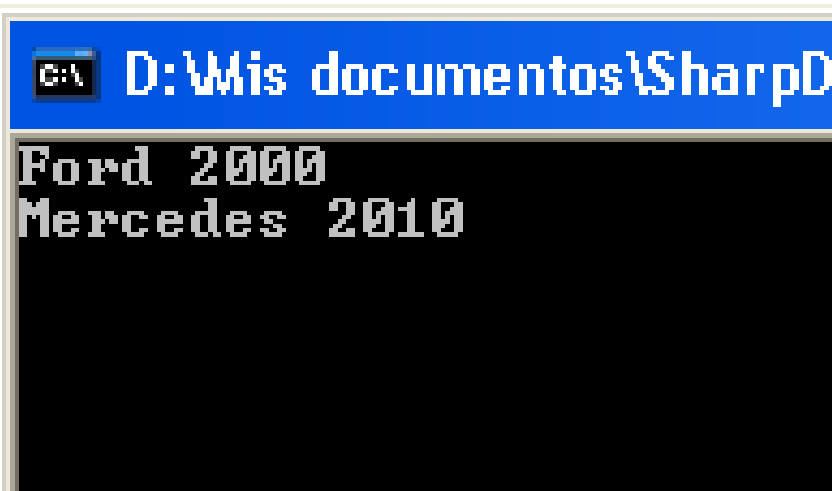
```
class Auto{  
    public string Marca;  
    public int Modelo;  
    public tipoAuto Tipo;  
    public void Imprimir(){  
        Console.WriteLine("{0} {1}",Marca, Modelo);  
    }  
}
```

```
class Colectivo{  
    public string Marca;  
    public int Modelo;  
    public int CantPasajeros;  
    public void Imprimir(){  
        Console.WriteLine("{0} {1}",Marca, Modelo);  
    }  
}
```

Derivación de clases

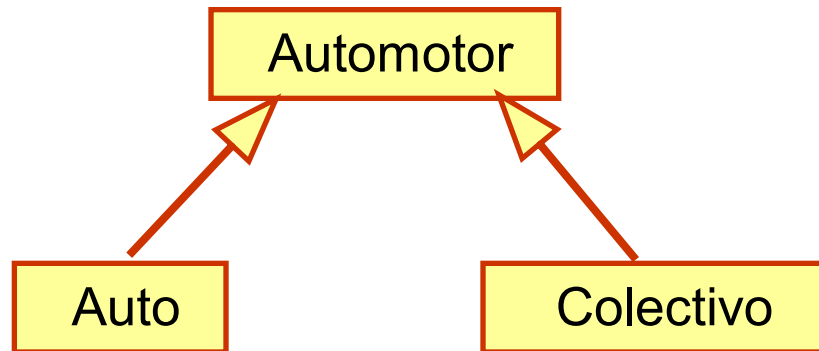
```
using System;
class programa{
    public static void Main(){
        Auto a=new Auto();
        Colectivo c=new Colectivo();
        a.Marca="Ford"; a.Modelo=2000;
        c.Marca="Mercedes"; c.Modelo=2010;
        c.CantPasajeros=20; a.Tipo=tipoAuto.Deportivo;
        a.Imprimir();
        c.Imprimir();
        Console.ReadKey();
    }
}
```

Ejecute y compruebe
su funcionamiento



Derivación de clases

- Claramente las clases **Auto** y **Colectivo** comparten tanto estructura 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”

Derivación de clases

Modifique el código de la siguiente manera (la clase Programa no se modifica)

```
class Automotor{  
    public string Marca;  
    public int Modelo;  
    public void Imprimir(){  
        Console.WriteLine("{0} {1}", Marca, Modelo);  
    }  
}
```

Auto deriva de Automotor

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

Colectivo deriva de Automotor

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

Ejecute y compruebe
que sigue funcionando
de la misma forma

Derivación de clases

Nota: Todas las clases en la plataforma .NET derivan directa o indirectamente de la clase **System.Object**

Por lo tanto en el código anterior:

```
class Automotor{  
    ...  
}
```

Es equivalente a:

```
class Automotor:Object{  
    ...  
}
```

Herencia

- Las clases **Auto** y **Colectivo** derivan de la clase **Automotor**, por lo tanto un **Auto** es un **Automotor** y un **Colectivo** también es un **Automotor**.
- Las variables **Marca** y **Modelo** definidas en la clase **Automotor**, son heredadas 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;
```

Herencia

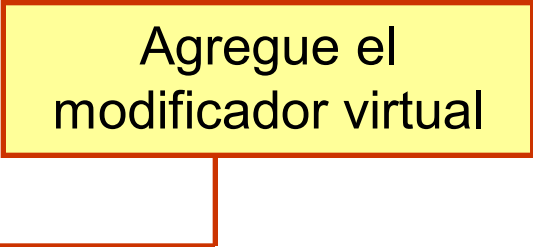
- Las clases **Auto** y **Colectivo** también heredan el método **imprimir()** definido en **Automotor**.
- Sin embargo, el método **imprimir()** resulta poco útil al no poder acceder a las variables específicas de **Auto** y **Colectivo** (**Tipo** y **CantPasajeros** respectivamente)
- **La solución:**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

Redefinición de métodos

- Primero se debe indicar en la clase **Automotor** que el método **imprimir()** es un método **virtual**. Esto permite su redefinición en las subclases (de lo contrario, en las subclases no se estaría redefiniendo sino ocultando el método imprimir de la superclase)

```
class Automotor{  
    public string Marca;  
    public int Modelo;  
    public virtual void Imprimir() {  
        Console.WriteLine("{0} {1}", Marca, Modelo);  
    }  
}
```

Agregue el
modificador virtual



Redefinición de métodos

Modifique la definición de las clases **Auto** y **Colectivo**

```
class Auto:Automotor{  
    public tipoAuto Tipo;  
    public override void Imprimir() {  
        Console.WriteLine("Auto {0} {1} {2}",  
                           Tipo, Marca, Modelo);  
    }  
}
```

El modificador override indica que se trata de una redefinición

Redefinición de métodos

Modifique la definición de las clases **Auto** y **Colectivo**

```
class Colectivo:Automotor{  
    public int CantPasajeros;  
    public override void Imprimir() {  
        Console.WriteLine("Colectivo {0} {1} {2} pasajeros ",  
                           Marca, Modelo, CantPasajeros);  
    }  
}
```

Ejecute y compruebe su funcionamiento



```
C:\Documents and Settings\Administrador\Mis documentos  
Auto Deportivo Ford 2000  
Colectivo Mercedes 2010 20 pasajeros
```

Acceso a miembros de la superclase

- 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 de la clase base
 - Para indicar a qué constructor de la clase base se debe llamar al crear instancias de la clase derivada.

Acceso a miembros de la superclase

- Utilización de la palabra clave **base** para invocar un método de la superclase. Modifique el método `imprimir()` de la clase **Auto** de la siguiente manera.

```
public override void Imprimir() {  
    Console.Write("Auto {0} ", Tipo);  
    base.Imprimir();  
}
```

Constructores - Revisitado

- Agregue el siguiente constructor a la clase **Automotor**

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

¿Qué sucede al
compilar?

Constructores - Revisitado

- Agregue el siguiente constructor a la clase **Automotor**

```
public Automotor () {  
    }  
}
```

- Y ahora...¿Por qué compila?

Constructores - Revisitado


Respuesta: 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

```
class Auto:Automotor{  
    public Auto():base(){  
    }  
    . . .  
}
```

```
class Colectivo:Automotor{  
    public Colectivo():base(){  
    }  
    . . .  
}
```

Constructores por defecto agregados automáticamente por el compilador

:base() llama al constructor sin argumentos de la clase base.



Constructores - Revisitado

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

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

Constructor en clase **Auto**

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

Constructor en
clase **Colectivo**

Constructores - Revisitado

Modifique el método **Main** de la clase **Programa** de la siguiente forma:

```
using System;
class programa{
    public static void Main() {
        Auto a=new Auto("Ford",2000,tipoAuto.Deportivo);
        Colectivo c=new Colectivo("Mercedes",2010,20);
        a.Imprimir();
        c.Imprimir();
        Console.ReadKey();
    }
}
```

Modificadores de acceso

Establezca las variables **Marca** y **Modelo** de la clase **Automotor** como privadas reemplazando el modificador **public** por **private**

¿Qué sucede? ¿Por qué no compila?

Respuesta: En el método **imprimir()** de la clase **Colectivo** intenta leer las variables **Marca** y **Modelo** que ahora son privadas de la clase **Automotor**, y por lo tanto, no accesibles desde el código de ninguna otra clase.

```
public override void Imprimir() {  
    Console.WriteLine("Colectivo {0} {1} {2} pasajeros ",  
                      Marca, Modelo, CantPasajeros);  
}
```

Variables inaccesibles debido
a su nivel de protección

Modificadores de acceso

Solución: Establezca las variables **Marca** y **Modelo** de la clase **Automotor** como protegidas reemplazando el modificador **private** por **protected**.

Los miembros protegidos sólo pueden accederse desde la propia clase en la que se definieron y desde todas las clases derivadas de la misma.

Modificadores de acceso

- Los **miembros de las clases** pueden declararse como públicos, internos, protegidos o privados.
 - Los miembros públicos son precedidos por el modificador de acceso **public**.
 - Los miembros internos son precedidos por el modificador de acceso **internal**.
 - Los miembros protegidos son precedidos por el modificador de acceso **protected**.
 - Los miembros privados son precedidos por el modificador de acceso **privated** (por defecto).

Modificadores de acceso

- Los **miembros públicos** pueden accederse desde cualquier clase de cualquier ensamblado que compone la aplicación.
- Los **miembros internos**, sólo desde las clases en el mismo ensamblado.
- Los **miembros protegido** sólo desde la propia clase o desde sus clases derivadas.
- Los **miembros privados** sólo desde la propia clase.

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

Modificadores de acceso

- Al combinar las palabras clave **protected** e **internal**, un miembro se declara como protegido e interno al mismo tiempo. Estos miembros pueden ser accedidos desde cualquier clase dentro del mismo ensamblado y desde sus clases derivadas definidas en cualquier ensamblado

Modificadores de acceso

- Las **clases** pueden declararse como **públicas** o **internas**.
 - Las clases **públicas** son precedidas por el modificador de acceso **public**.
 - Las clases **internas** (valor predeterminado) 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

Polimorfismo

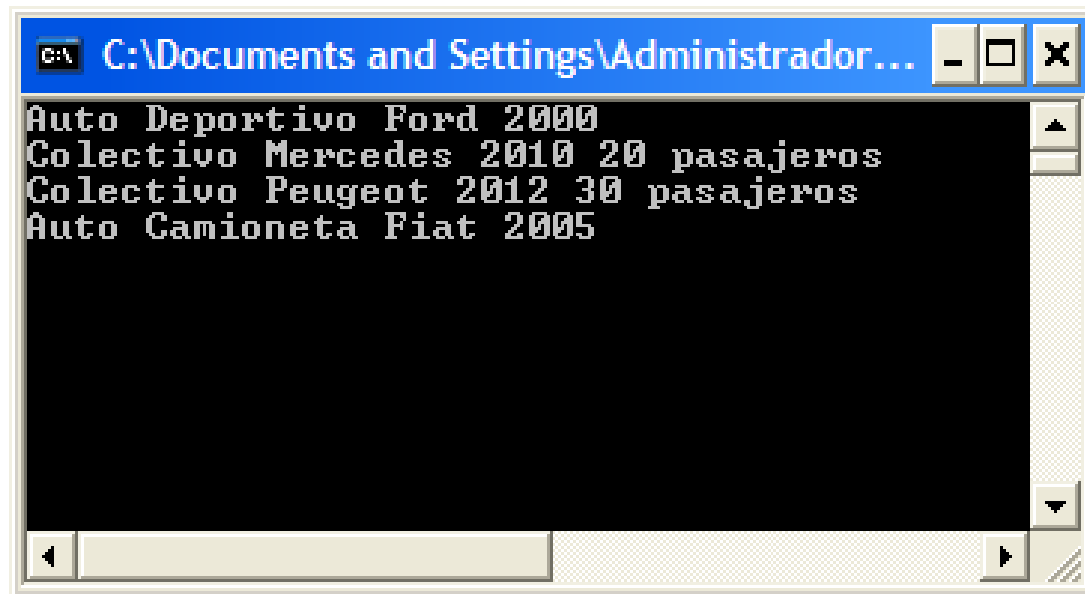
Modifique el método Main de la siguiente forma

```
public static void Main() {  
    Automotor[] vector=new Automotor[4];  
    vector[0] = new Auto("Ford",2000,tipoAuto.Deportivo);  
    vector[1] = new Colectivo("Mercedes",2010,20);  
    vector[2] = new Colectivo("Peugeot",2012,30);  
    vector[3] = new Auto("Fiat",2005,tipoAuto.Camioneta);  
    foreach(Automotor a in vector){  
        a.Imprimir();  
    }  
    Console.ReadKey();  
}
```

Ejecute y compruebe su
funcionamiento

Polimorfismo

El código anterior produce la siguiente salida:



```
C:\Documents and Settings\Administrador...  
Auto Deportivo Ford 2000  
Colectivo Mercedes 2010 20 pasajeros  
Colectivo Peugeot 2012 30 pasajeros  
Auto Camioneta Fiat 2005
```

Observe que un mismo mensaje (imprimir) enviado a objetos distintos produce resultados diferentes (los autos y los colectivos se imprimen de distinta manera en la consola). Justamente a esto se denomina **polimorfismo**

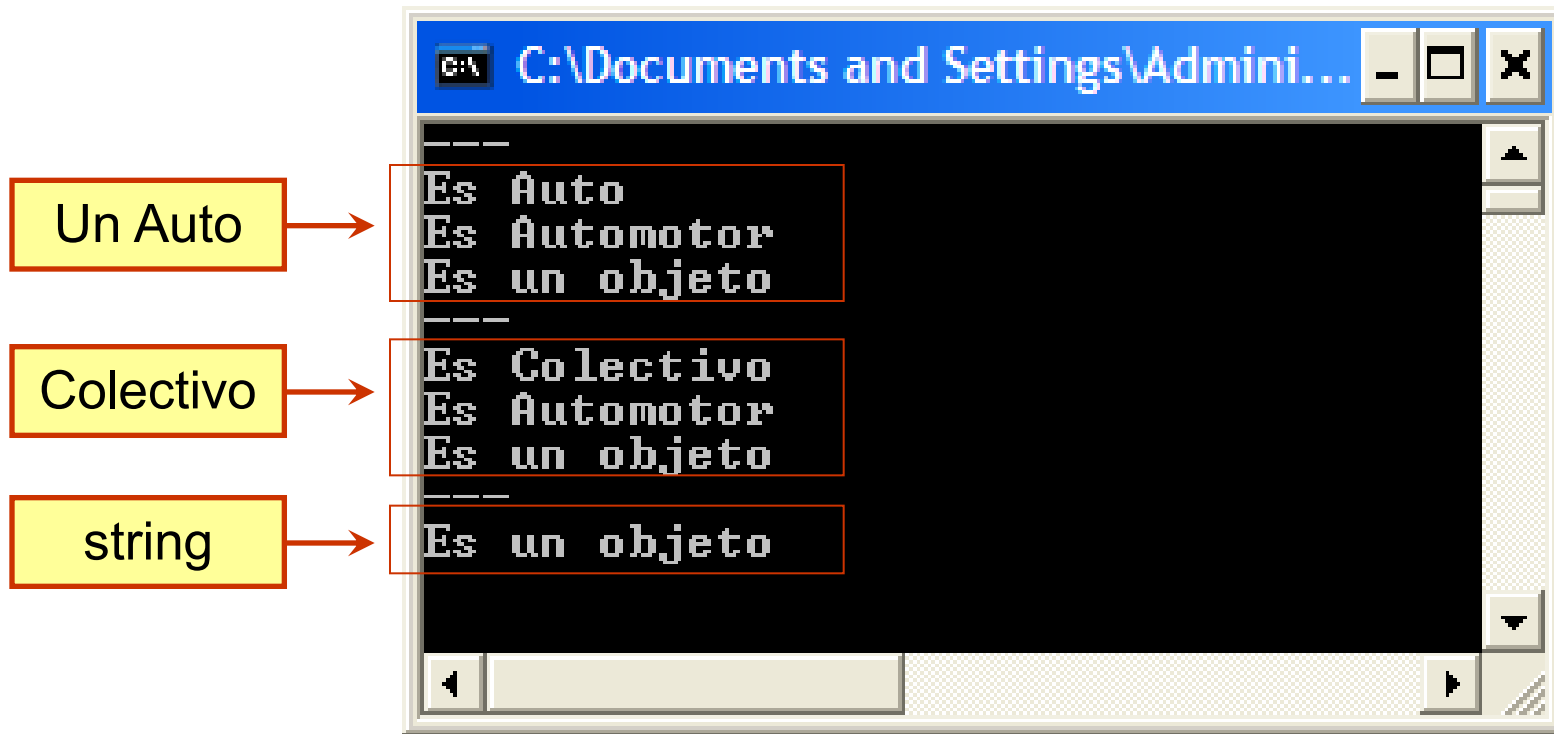
Operador is

Consultar el tipo de una variable

<expresión> is <nombreTipo>

```
using System;
class programa{
    public static void Main(){
        queEs(new Auto("Fiat",2000, tipoAuto.Familiar));
        queEs(new Colectivo("Mercedez",2000,20));
        queEs("CASA");
        Console.ReadKey();
    }
    static void queEs(object o){
        Console.WriteLine("---");
        if (o is Auto) Console.WriteLine("Es Auto");
        if (o is Colectivo) Console.WriteLine("Es Colectivo");
        if (o is Automotor) Console.WriteLine("Es Automotor");
        if (o is object) Console.WriteLine("Es un objeto");
    }
}
```

Operador is



Observe que un Auto también es Automotor, un Colectivo también es Automotor, y que todos (incluido el string) son también objetos

Operador as (revisitado)

<expresion> as <tipoDestino>

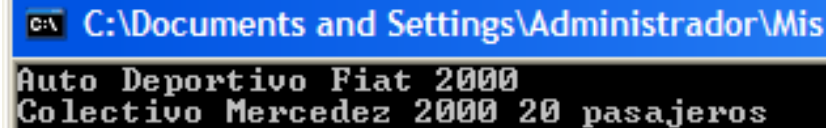
- A veces se utiliza conjuntamente con el operador **is**.
- Con el operador **is** se determina el tipo de objeto y luego con el operador **as** se convierte al tipo apropiado.

Operador as (revisitado)

```
using System;
using System.Collections;
class programa{
    public static void Main(){
        ArrayList lista=new ArrayList();
        lista.Add(new Auto("Fiat",2000,tipoAuto.Deportivo));
        lista.Add("CASA");
        lista.Add(78.5);
        lista.Add(new Colectivo("Mercedez",2000,20));

        foreach(object o in lista){
            if (o is Automotor){
                (o as Automotor).Imprimir();
            }
        }
        Console.ReadKey();
    }
}
```

Sólo si o es un Automotor
se convierte e se invoca su
método Imprimir()



```
C:\Documents and Settings\Administrador\Mis...
Auto Deportivo Fiat 2000
Colectivo Mercedez 2000 20 pasajeros
```

Destruyctores

- Gestión de destrucción de objetos
- Liberación de recursos
- Puede haber un solo destructor por clase, no puede tener parámetros ni modificadores

```
~<nombreTipo>()  
{  
    <código>  
}
```

- Siempre se llama al del tipo base
- El compilador define uno

Destruyores

Codifique los destructores de las clases **Automotor** y **Colectivo** de la siguiente manera:

```
~Automotor() {  
    Console.WriteLine("Destructor Automotor");  
}
```

```
~Colectivo() {  
    Console.WriteLine("Destructor Colectivo");  
}
```


Destructores

```
using System;
class programa{
    static Colectivo c;
    public static void Main(){
        c=new Colectivo("Mercedes",2011,40);
        Console.WriteLine("Se instanci6");
        c=null;
        Console.WriteLine("Se desreferenci6");
        Console.ReadKey();
    }
}
```

Ejecute y compruebe su funcionamiento

C:\Documents and Settings\Administrado

Se instanci6
Se desreferenci6
—

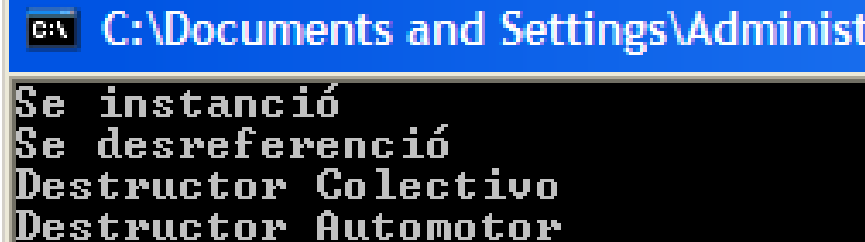
Garbage Collection

- El garbage collector (**GC**) es un servicio del **CLR** que remueve automáticamente los objetos de la memoria (destruye) cuando se convierten en inaccesibles desde el programa.
- El **GC** trabaja de manera asincrónicamente, sin embargo si se desea realizar la recolección en un determinado momento puede utilizarse el método estático `GC.Collect()`

Garbage Collection

```
using System;
class programa{
    static Colectivo c;
    public static void Main(){
        c=new Colectivo("Mercedes",2011,40);
        Console.WriteLine("Se instanci6");
        c=null;
        GC.Collect();
        Console.WriteLine("Se desreferenci6");
        Console.ReadKey();
    }
}
```

Agregue esta instrucci3n
y vuelva a ejecutar



```
C:\Documents and Settings\Administ
Se instanci6
Se desreferenci6
Destructor Colectivo
Destructor Automotor
```