

C#

Clases - Continuación

Derivación de clases

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

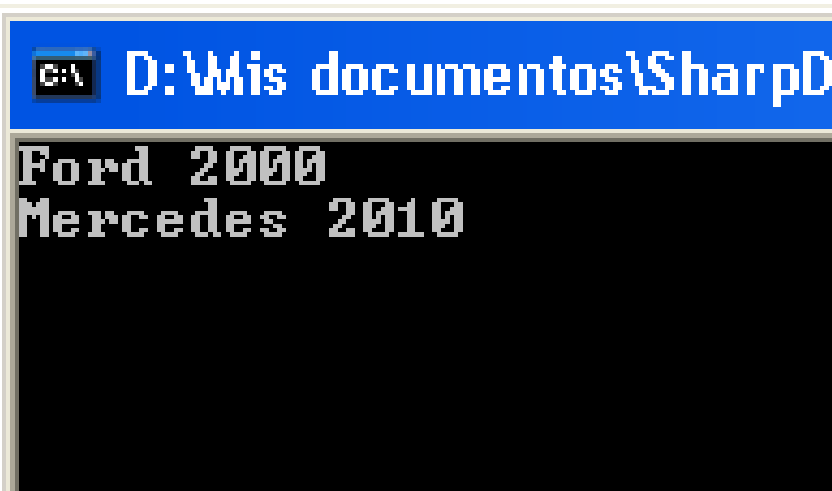
```
class Auto{  
    public string Marca;  
    public int Modelo;  
    public tipoAuto Tipo;  
    public void Imprimir(){  
        Console.WriteLine("{0} {1}",Marca, Modelo);  
    }  
}
```

```
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

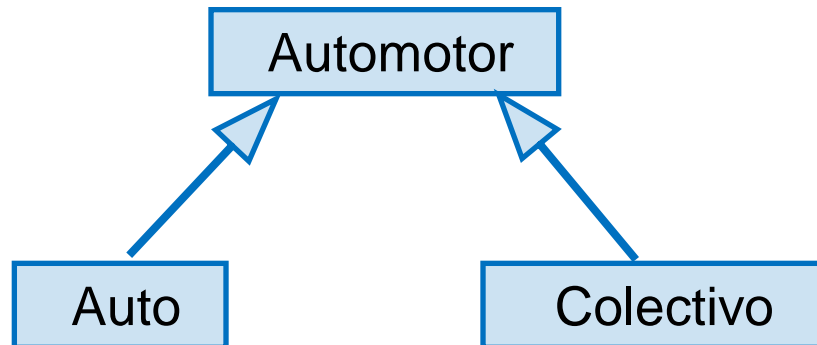


The screenshot shows a Windows command prompt window with a blue title bar. The title bar text is "D:\Mis documentos\SharpD". The command prompt window has a black background with white text. The output of the program is displayed as two lines: "Ford 2000" and "Mercedes 2010".

```
D:\Mis documentos\SharpD
Ford 2000
Mercedes 2010
```

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

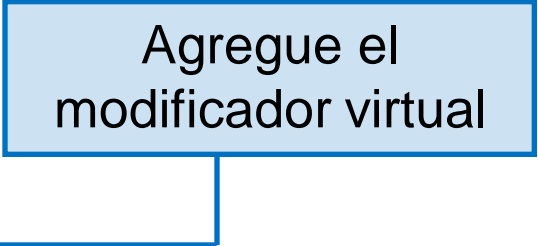
- Las clases **Auto** y **Colectivo** también heredan el método **imprimir()** definido en **Automotor**.
- Sin embargo, el método **imprimir()** resulta poco útil al no poder acceder a las variables específicas de **Auto** y **Colectivo** (**Tipo** y **CantPasajeros** respectivamente)
- **La solución:** 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

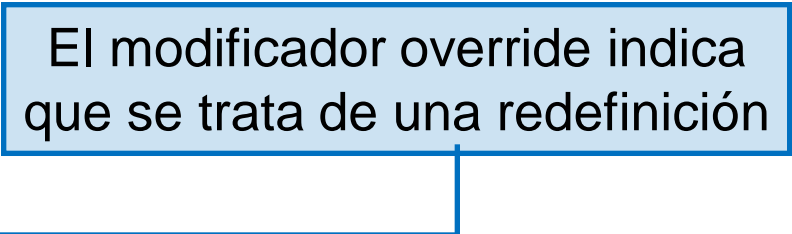


Redefinición de métodos

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.WriteLine("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 **private** (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

Modificadores de accesos

Las propiedades pueden tener distinto nivel de acceso para lectura que para escritura

```
class Persona {  
    private string nombre;  
    public string Nombre  
    {  
        get  
        {  
            return nombre;  
        }  
        protected set  
        {  
            nombre = value;  
        }  
    }  
}
```

Se le define la propiedad con el nivel de acceso más amplio y se restringe la sección **get** o **set** con un nivel de acceso más restrictivo

Modificadores de accesos

Codifique y compile

```
class Persona  
{  
    public string Nombre { get; set; }  
}
```

Por defecto las
clases son **internal**

```
public class Auto  
{  
    private Persona dueño1;  
    protected Persona dueño2;  
    internal Persona dueño3;  
    public Persona dueño4;  
}
```

Error de compilación:
El tipo de un campo debe
ser al menos tan accesible
como el propio campo.

Modificadores de accesos

```
class Persona {  
    public string Nombre { get; set; }  
}
```

¿Existe algún problema con este código?

```
public class Auto {  
    private Persona dueño1, dueño2, dueño3, dueño4;  
  
    public Persona GetPrimerDueño() {  
        return dueño1;  
    }  
  
    protected Persona UltimoDueño {  
        set {  
            dueño4 = value  
        }  
    }  
}
```

Error de compilación:

El tipo **Persona** es menos accesible que el tipo de retorno del método **GetPrimerDueño()** y del tipo de la propiedad **UltimoDueño**

Restricciones en el uso de los niveles de accesibilidad

- La clase base directa de un tipo de clase debe ser al menos tan accesible como el propio tipo de clase.

```
class Persona  
{  
}
```

**Error de
compilación**

```
public class Empleado: Persona  
{  
}
```

Restricciones en el uso de los niveles de accesibilidad

- El tipo de las constantes, campos y propiedades debe ser al menos tan accesible como la propia constante, campo o propiedad
- El tipo de valor devuelto y los tipos de parámetros de un método deben ser al menos tan accesibles como el propio método.
- El tipo y los tipos de parámetros de un indizador deben ser al menos tan accesibles como el propio indizador.
- Los tipos de parámetros de un constructor deben ser al menos tan accesibles como el propio constructor.

La privacidad es a nivel de clases, no de objetos

```
class Animal {  
    private string nombre;  
    private 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

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

Válido

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

Pregunta sobre herencia

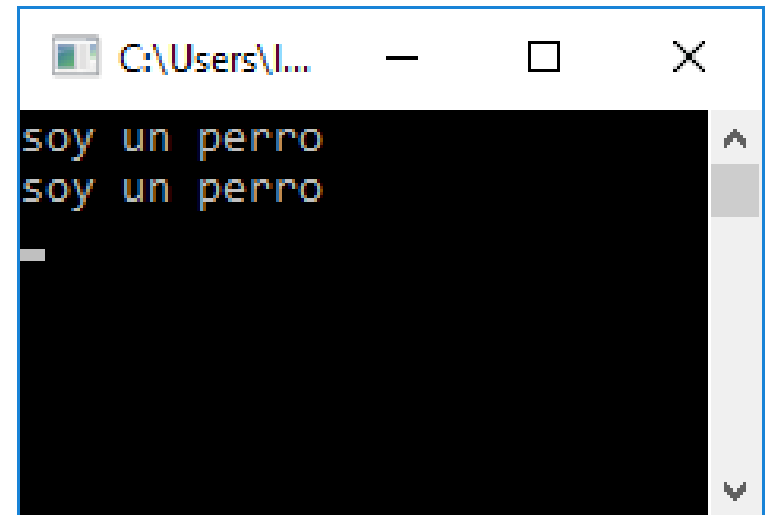
**¿Se heredan
los campos estáticos?**

**Proponga un ejemplo de código para
comprobarlo**

```
class Program
{
    public static void Main(string[] args) {
        Animal.Nombre = "soy un animal";
        Perro.Nombre = "soy un perro";
        System.Console.WriteLine(Animal.Nombre);
        System.Console.WriteLine(Perro.Nombre);
        System.Console.ReadKey();
    }
}
```

```
class Animal {
    public static string Nombre;
}
```

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



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\I...' and standard window controls. The command prompt has a black background with white text. It displays two lines of output: 'soy un perro' on the first line and 'soy un perro' on the second line. A white cursor is visible on the third line.

Pregunta sobre herencia

¿Se heredan
los campos estáticos?

Rta.: NO

En el código anterior la clase **Perro** no heredó el campo estático **Nombre** de la clase **Animal**.

Simplemente se accedió al campo estático de la clase **Animal** a través de la clase derivada **Perro** 33

Ocultamiento de Campos estáticos en las subclases

- Es posible **volver a definir un campo estático** en una subclase.
- De esta forma se dice que el nuevo campo **oculta** al campo de la clase base

```

class Program
{
    public static void Main(string[] args) {
        Animal.Nombre = "soy un animal";
        Perro.Nombre = "soy un perro";
        System.Console.WriteLine(Animal.Nombre);
        System.Console.WriteLine(Perro.Nombre);
        System.Console.ReadKey();
    }
}

```

```

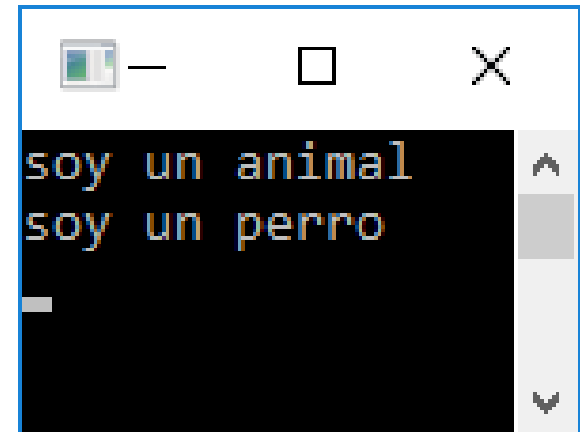
class Animal {
    public static string Nombre;
}

```

```

class Perro: Animal {
    public new static string Nombre;
}

```



Con **new** se evita el *warning* del compilador que nos advierte que estamos ocultando el campo **Animal.Nombre**

Herencia de métodos estáticos

Los **métodos estáticos no pueden redefinirse** en las subclases. El código siguiente no compila.

```
class Program
{
    public static void Main(string[] args) {
        Animal.MostrarInformacion();
        Perro.MostrarInformacion();
        System.Console.ReadKey();
    }
}
```

Error de compilación: Un miembro estático 'Animal.MostrarInformacion()' no se puede marcar como override, virtual o abstract

```
class Animal {
    public virtual static void MostrarInformacion()
    {
        System.Console.WriteLine("clase Animal");
    }
}
```

Error de compilación: Un miembro estático 'Perro.MostrarInformacion()' no se puede marcar como override, virtual o abstract

```
class Perro: Animal {
    public override static void MostrarInformacion()
    {
        System.Console.WriteLine("clase Perro");
    }
}
```

Herencia de métodos estáticos

Igualmente al caso de los campos, los **métodos estáticos pueden ser ocultos** en las subclases.

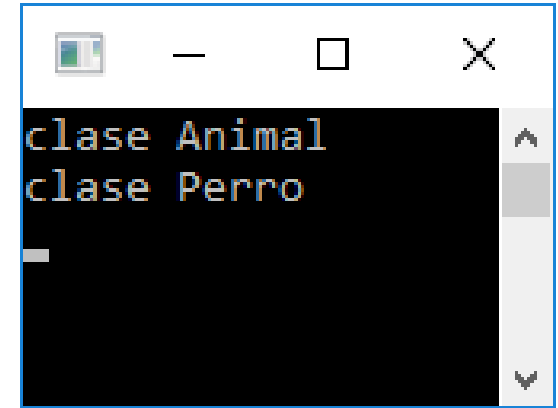
```

class Program
{
    public static void Main(string[] args) {
        Animal.MostrarInformacion();
        Perro.MostrarInformacion();
        System.Console.ReadKey();
    }
}

class Animal {
    public static void MostrarInformacion()
    {
        System.Console.WriteLine("clase Animal");
    }
}

class Perro: Animal {
    public new static void MostrarInformacion()
    {
        System.Console.WriteLine("clase Perro");
    }
}

```



```

clase Animal
clase Perro

```

Con **new** se evita el *warning* del compilador que nos advierte que estamos ocultando miembro **Animal.MostrarInformacion()**

Clases abstractas

- En algunos casos es conveniente declarar clases para las cuales nunca se crearán instancias de objetos. A dichas clases se les conoce como **clases abstractas**.
- El propósito principal de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común.
- La clase **Automotor** de los ejemplos anteriores podría haberse definido como abstracta.
- Las clases abstractas se modifican con la palabra clave ***abstract***

Clases abstractas

- Una clase abstracta puede tener métodos abstractos y concretos.
- Los métodos abstractos se escriben sin el cuerpo, alcanza con marcar el encabezado con la palabra clave ***abstract***
- Si una clase que posee un método abstracto no es declarada como abstracta, el compilador provoca un error de compilación.

Ejemplo Clase abstracta

Clase abstracta

`abstract class Automotor`

{

`protected string Marca;`

`protected int Modelo;`

`public abstract void Imprimir();`

}

No hay implementación aquí

Tratar de instanciar un objeto de una clase abstracta produce un error de compilación.

Si no se implementan los métodos y las propiedades abstractas de la clase base en una clase derivada, se produce un error de compilación, a menos que la clase derivada también se declare como **abstract**

Polimorfismo

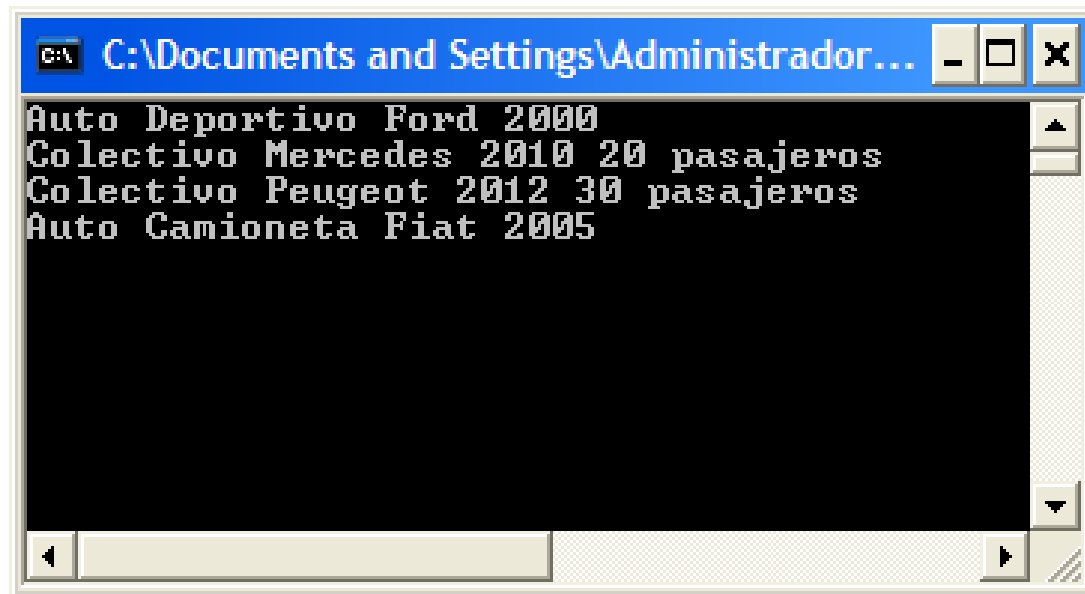
Modifique el método Main de la siguiente forma

```
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**

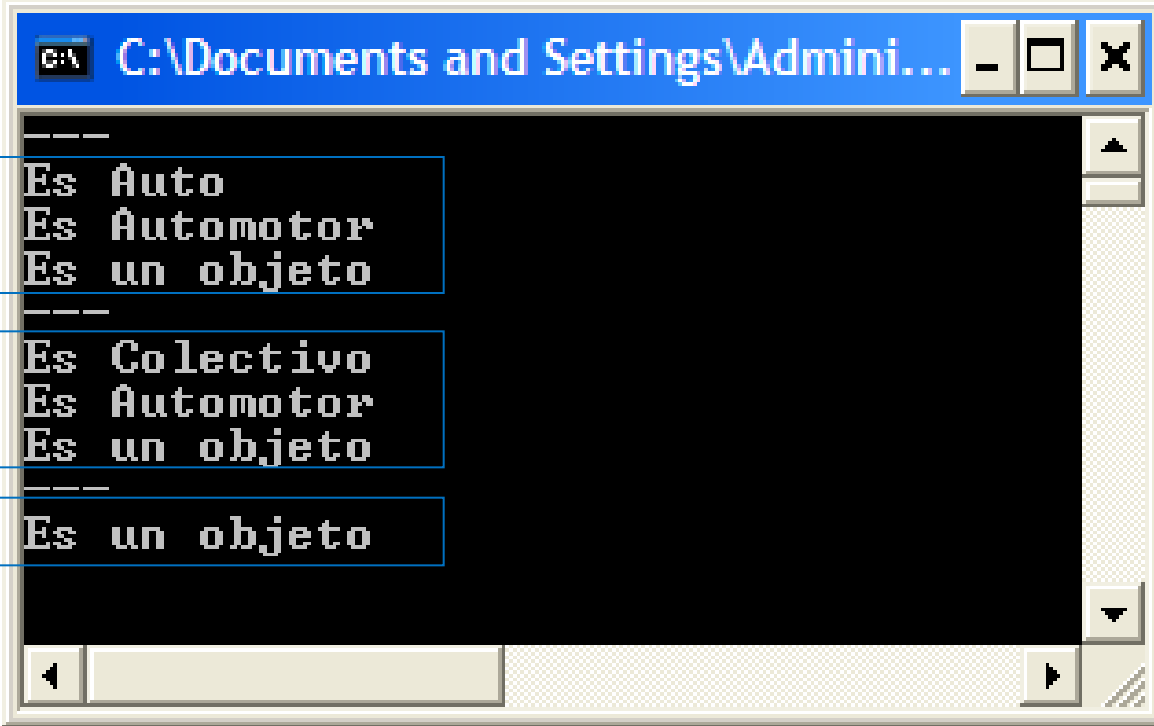
Operador is

Consultar el tipo de una variable

<expresión> is <nombreTipo>

```
using System;
class programa{
    public static void Main(){
        queEs(new Auto("Fiat",2000,tiposAuto.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



The screenshot shows a Windows command prompt window with the title bar 'C:\Documents and Settings\Admini...'. The command prompt displays the following output:

```
---  
Es Auto  
Es Automotor  
Es un objeto  
---  
Es Colectivo  
Es Automotor  
Es un objeto  
---  
Es un objeto
```

Three blue boxes on the left side of the image point to the corresponding lines of output:

- 'Un Auto' points to 'Es Auto'.
- 'Colectivo' points to 'Es Colectivo'.
- 'string' points to 'Es un objeto'.

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>

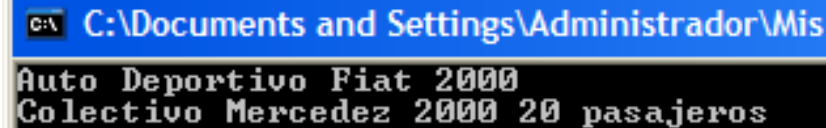
- A veces se utiliza conjuntamente con el operador **is**.
- Con el operador **is** se determina el tipo de objeto y luego con el operador **as** se convierte al tipo apropiado.

Operador as (revisitado)

```
using System;
using System.Collections;
class programa{
    public static void Main(){
        ArrayList lista=new ArrayList();
        lista.Add(new Auto("Fiat",2000,tipoSistema.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
```


Destructores

- 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
—

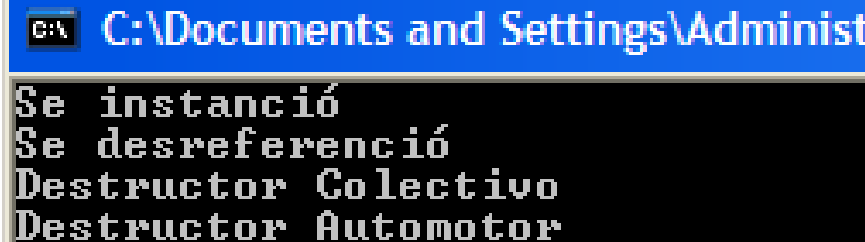
Garbage Collection

- 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
```

Seminario .NET

Autoevaluación 2

1) ¿Cuál de las siguientes afirmaciones es verdadera?

0) No sabe / No contesta

1) La BCL ofrece a las aplicaciones un entorno de ejecución administrado y servicios automáticos como la administración de memoria.

→ 2) El CLR define un entorno de ejecución en el que trabajan las aplicaciones escritas con cualquier lenguaje .NET

3) El CLR es una especie de código ensamblador pero de más alto nivel

2) Qué imprime en la consola el siguiente fragmento de código

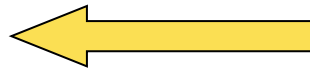
```
double d=2; int i=2;  
Console.Write(3/i + " ");  
Console.Write(3/d + " ");  
Console.Write(3/2.0 + " ");  
Console.Write(3/2 + " ");
```

0) No sabe / No contesta

1) 1 1 1,5 1

2) 1,5 1,5 1,5 1

3) 1 1,5 1,5 1



3) Indique cuál de las siguientes afirmaciones es FALSA

0) No sabe / No contesta

1) El tipo object es una estructura 

2) Los tipos flotantes (float y double) son tipos valor

3) Las clases, delegados e Interfaces son tipos referencias

4) ¿Qué salida por consola produce el siguiente código?

```
public static void Main(string[] args)
{
    try{
        metodo(0);
        Console.Write("A");
    }catch{
        Console.Write("B");
    }
}
static void metodo(int x){
    try{
        int y=1/x;
    }finally{
        Console.Write("C");
    }
}
```

0) No sabe / No contesta

1) **CB** 

2) **CA**

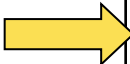
3) **CAB**

6) Indique cual de todas las afirmaciones es verdadera respecto de las siguientes definiciones de clases.

```
class Auto{  
    private string marca;  
    public Auto(string marca) {  
        this.marca=marca;  
    }  
}  
class Taxi:Auto{  
}
```

0) No sabe / No contesta

1) Aunque compilen correctamente ambas clases, nunca podrá instanciarse un objeto de clase `Taxi`

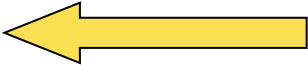
 2) La definición de la clase `Taxi` provocará un error de compilación

3) Ambas clases se compilan sin errores

5) Indique cual de todas las afirmaciones es verdadera respecto del siguiente programa

```
using System;
class Program{
    static void Main(){
        Auto a;
        a.Marca = "Ford";
        Console.WriteLine(a.Modelo);
        Console.ReadKey();
    }
}
class Auto{
    public string Marca;
    public int Modelo;
}
```


0) No sabe / No contesta

- 1) La línea `a.Marca = "Ford";` provoca un error en tiempo de ejecución (excepción: `NullReferenceException`)
- 2) La línea `Console.WriteLine(a.Modelo);` provoca un error en tiempo de compilación (variable `Modelo` no asignada)
- 3) La línea `a.Marca = "Ford";` provoca un error en tiempo de compilación (variable local no asignada) 

6) Indique cual de todas las afirmaciones es verdadera respecto del siguiente programa

```
using System;
class Program{
    static Auto a;
    static void Main(){
        a.Marca = "Ford";
        Console.WriteLine(a.Modelo);
        Console.ReadKey();
    }
}
class Auto{
    public string Marca;
    public int Modelo;
}
```

0) No sabe / No contesta

1) La línea `a.Marca = "Ford";` provoca un error en tiempo de ejecución (excepción: `NullReferenceException`) 

2) La línea `Console.WriteLine(a.Modelo);` provoca un error en tiempo de compilación (variable `Modelo` no asignada)

3) La línea `a.Marca = "Ford";` provoca un error en tiempo de compilación (variable local no asignada)

- Recordar que las variables locales no son inicializadas automáticamente pero las variables de instancia y las estáticas sí lo son.
- Respecto de las variables locales, el compilador es capaz de detectar si existe un intento de leer una variable local no inicializada. En este caso, el error producido es de compilación.
- Respecto de las variables de instancia y estáticas, siempre pueden ser leídas aún sin haber sido inicializadas explícitamente. Éstas son inicializadas con el valor por defecto según su tipo. Los números se inicializan en 0, los objetos en **null**.