UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

INTRODUCCION A LA PROGRAMACION C#

3 - MÓDULO 1 : FUNDAMENTOS DE LA PROGRAMACION ORIENTADA A OBJETOS

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

UNIDAD: 3

MODULO: 1

PRESENTACIÓN: En esta unidad se explicarán los fundamentos de la programación orientada a objetos.

OBJETIVOS

Que los participantes logren: Comprender los pilares de la POO y su aplicación en C#

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

TEMARIO

Clases y objetos

Qué son las clases, qué son los objetos y en qué se diferencian ?

La clase es la "plantilla" en la que nos basamos para crear el objeto.

Pongamos el ejemplo de los coches, todos ellos tienen una serie de características comunes:

- Todos tienen un motor, ruedas, un volante, pedales, chasis, carrocería...
- Todos funcionan de un modo parecido para acelerar, frenar, meter las marchas, dar las luces, etc.

Sin embargo, cada uno de ellos es diferente de los demás, puesto que cada uno es de su marca, modelo, color, número de patente..., propiedades que lo diferencian de los demás, aunque una o varias de ellas puedan coincidir en varios coches.

Diríamos entonces que todos los coches están basados en una plantilla, o un tipo de objeto, es decir, pertenecen todos a la misma clase: **la clase coche**.

Sin embargo, cada uno de los coches es un objeto de esa clase: todos comparten la "interfaz", pero no tienen por qué compartir los datos (marca, modelo, color, etc).

Se dice entonces que cada uno de los objetos es una instancia de la clase a la que pertenece.

En resumen, la clase es algo genérico (la idea que todos tenemos sobre lo que es un coche) y el objeto es algo mucho más concreto (el coche del vecino, el nuestro, el batimovil).

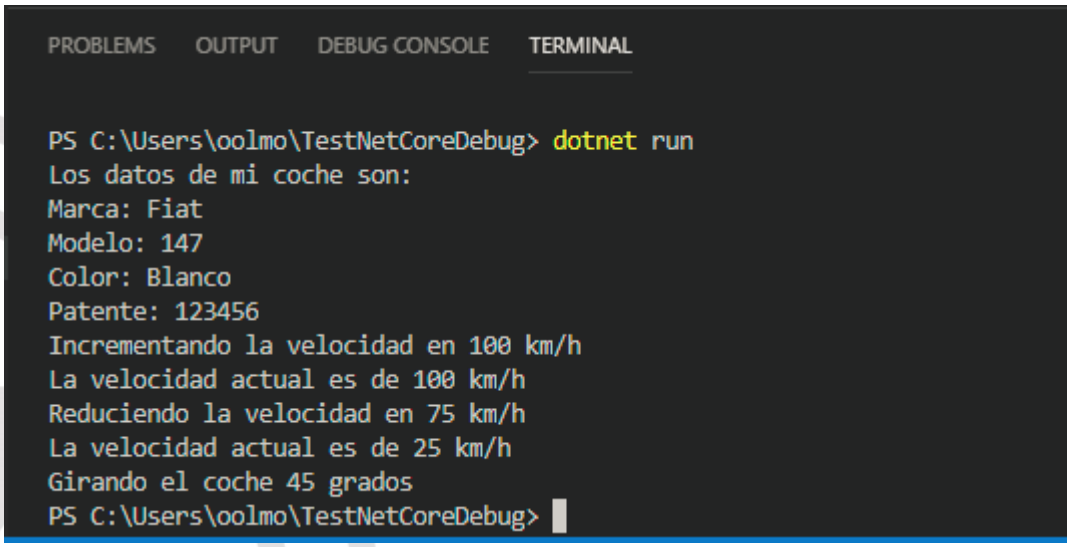
El diseño de la clase Coche sería algo parecido a esto (aunque más ampliado):

```
using System;
namespace OOP
{
    class Coche
    {
        public double velocidad = 0;
        public string marca;
        public string modelo;
        public string color;
        public string patente;

        public Coche(string marca, string modelo, string color, string patente)
        {
            this.marca = marca;
            this.modelo = modelo;
            this.color = color;
            this.patente = patente;
        }
        public void Acelerar(double cantidad)
        {
            // Aquí se le dice al motor que aumente las revoluciones
            Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);
            this.velocidad += cantidad;
        }
        public void Girar(double cantidad)
        {
            // Aquí iría el código para girar
            Console.WriteLine("Girando el coche {0} grados", cantidad);
        }
        public void Frenar(double cantidad)
        {
            // Aquí se le dice a los frenos que actúen, y...
            Console.WriteLine("Reduciendo la velocidad en {0} km/h", cantidad);
            this.velocidad -= cantidad;
        }
    }
}
```

Veamos una clase con un método *Main* para ver cómo se comportaría esta clase:

```
using System;


namespace OOP
{
class Program
{
    static void Main()
    {
        Coche miCoche=new Coche("Fiat", "147", "Blanco","123456");
        Console.WriteLine("Los datos de mi coche son:");
        Console.WriteLine("Marca: {0}", miCoche.marca);
        Console.WriteLine("Modelo: {0}", miCoche.modelo);
        Console.WriteLine("Color: {0}", miCoche.color);
        Console.WriteLine("Patente: {0}", miCoche.patente);

        miCoche.Acelerar(100);
        Console.WriteLine("La velocidad actual es de {0} km/h",miCoche.velocidad);
        miCoche.Frenar(75);
        Console.WriteLine("La velocidad actual es de {0} km/h",miCoche.velocidad);

        miCoche.Girar(45);

    }
}

}
```

El resultado que aparecería en la consola al ejecutar este programa sería este:



Conclusiones del ejemplo anterior:

- La clase es donde se definen todos los datos y se programan todas las acciones que han de manejar los objetos de esta clase.
- Los datos son *velocidad, marca, modelo, color y patente*
- Los métodos son *Acelerar, Girar y Frenar*.
- Para manejar *miCoche* (creado en la primera línea del método *Main*) uso la interfaz diseñada en la clase (la interfaz de una clase es el conjunto de métodos y propiedades que esta ofrece para su manejo).
- Coche es la clase y *miCoche* un objeto de esta clase.

Los pilares de la POO

Encapsulamiento

Podríamos definirlo como la capacidad que tienen los objetos de ocultar su código a quien hace uso de los mismos y proteger sus datos, ofreciendo única y exclusivamente una interfaz que garantiza que el uso del objeto es el adecuado.

La ocultación del código es algo evidente: cuando se invoca el método Acelerar del objeto miCoche, lo único que sabemos es que el coche acelerará, pero el cómo lo haga es algo que no podremos ver desde donde invocamos a este método del coche.

Las propiedades en C#

El ejemplo anterior tiene un error de modelado: Podemos modificar directamente el valor del campo velocidad, dado que es público.
La única forma de modificar su valor debería ser invocando los métodos Acelerar y/o Frenar. Esta importante característica aseguraría que los datos de los objetos pertenecientes a esta clase se van a manejar del modo adecuado.

Hagamos la siguiente modificación:

```
protected double velocidad=0;
public double Velocidad
{
get
    {
        return velocidad;
    }
}
```

Con lo cual ahora ejecutar en el ejemplo anterior lo siguiente, no sería valido.

```
miCoche.velocidad=100; // Esto provocaría un error. Velocidad es de sólo lectura
```

El único modo de modificar la velocidad del coche es mediante el método *Acelerar*

```
        miCoche.Acelerar(100);
        Console.WriteLine(miCoche.Velocidad);
```

Herencia

Otro de los pilares básicos de la POO es la herencia. Gracias a ella podemos definir clases nuevas basadas en clases antiguas, añadiéndoles más datos o más funcionalidad.

Para ver esto más claro sigamos con el ejemplo del coche. Imaginemos que la clase Coche ofrece una interfaz básica para cualquier tipo de coche. Sin embargo queremos un coche que, además de todo lo que tienen los demás coches, es capaz de estacionar él solito, sin necesidad de que nosotros andemos haciendo maniobras. ¿Tendríamos que definir otra clase para incorporar esta nueva capacidad? Pues no. Podemos heredar todos los miembros de la clase Coche y después agregarle lo que deseemos en la nueva clase:

```
using System;


namespace OOP
{
    class CocheInteligente:Coche
    {
        public CocheInteligente(string marca, string modelo, string color, string patente)
                                        : base(marca, modelo, color, patente) {}
        public void Estacionar()
        {
            // Aquí se escribe el código para que el coche estacione solo
            Console.WriteLine("Estacionando el coche de modo automático");
            velocidad = 0;
        }
    }
}
```

La clase *CocheInteligente* ha heredado todos los miembros de su clase base (*Coche*). Lo único que ha añadido ha sido el método Estacionar, de modo que cualquier objeto de la clase *CocheInteligente* (ojo,

no de la clase *Coche*) tendrá todos los miembros de la clase *Coche* más el método *Estacionar* incorporado en la clase derivada *CocheInteligente*. ¿Y cómo se instancian objetos de una clase derivada? Pues exactamente igual que si se instanciara de cualquier otra clase.

Veámoslo con el ejemplo anterior, modificando ligeramente el método *Main*:

```
using System;


namespace OOP
{
class Program
{
    static void Main()
    {
        CocheInteligente miCoche=new CocheInteligente("Peugeot", "306", "Azul","1546876");


        Console.WriteLine("Los datos de mi coche son:");
        Console.WriteLine("Marca: {0}", miCoche.marca);
        Console.WriteLine("Modelo: {0}", miCoche.modelo);
        Console.WriteLine("Color: {0}", miCoche.color);
        Console.WriteLine("Número de patente: {0}", miCoche.patente);


        miCoche.Acelerar(100);
        Console.WriteLine("La velocidad actual es de {0} km/h",miCoche.Velocidad);
        miCoche.Frenar(75);
        Console.WriteLine("La velocidad actual es de {0} km/h",miCoche.Velocidad);


        miCoche.Girar(45);
        miCoche.Estacionar();
        string a=Console.ReadLine();
    }
}
}
```

Ahora, el resultado en la consola sería este:

```
Los datos de mi coche son los siguientes:
Marca: Peugeot
Modelo: 306
Color: Azul
Número de patente: 1546876
Incrementando la velocidad en 100 km/h
La velocidad actual es de 100 km/h
Reduciendo la velocidad en 75 km/h
La velocidad actual es de 25 km/h
Girando el coche 45 grados
Aparcando el coche de modo automático
```

Ahora, el objeto *miCoche* tiene los mismos miembros que tenía cuando era de la clase Coche más el método Estacionar implementado por la clase derivada *CocheInteligente*.

Poder construir clases más complejas a partir de otras clases más sencillas es el objetivo principal de la herencia.

No obstante, C# soporta la herencia simple, pero no la herencia múltiple. Por lo tanto, en C# podemos construir una clase derivada a partir de otra clase, pero no de varias clases.

Sí se puede derivar una clase de otra clase y varias interfaces, pero de esto hablaremos más adelante, cuando tratemos las interfaces.

El polimorfismo es otra de las maravillas que incorpora la POO. ¿Qué ocurre si, siguiendo con este ejemplo de los coches, cada coche ha de comportarse de un modo diferente dependiendo de su marca, esto es, si es un Peugeot, por ejemplo, el acelerador acciona un cable, pero si es un Volkswagen, el acelerador acciona un mecanismo electrónico?.

Modelado Procedural (enfoque erróneo) :

```
public void Acelerar(double cantidad)
{
    switch (this.marca)
    {
        case "Peugeot":
            // Aquí acciona el mecanismo de aceleración de los Peugeot...
            Console.WriteLine("Accionando el mecanismo de aceleración del Peugeot");
            break;
        case "Volkswagen":
            // Aquí acciona el mecanismo de aceleración de los Volkswagen...
            Console.WriteLine("Accionando el mecanismo de aceleración del
                               Volkswagen");
            break;
        case "Seat":
            // Aquí acciona el mecanismo de aceleración de los Seat...
            Console.WriteLine("Accionando el mecanismo de aceleración del Seat");
            break;
        default:
            // Aquí acciona el mecanismo de aceleración por defecto...
            Console.WriteLine("Accionando el mecanismo de aceleración por defecto");
            break;
    }
    Console.WriteLine("Incrementando la velocidad en {0} km/h");
    this.velocidad += cantidad;
}
```

Este enfoque es erróneo porque, por cada marca que adopte una nueva forma de acelerar, vamos a tener que modificar el código de la clase *Coche*.

Polimorfismo

En un enfoque Orientado a Objetos, si nuestra clase *Coche* estuviese bien diseñada, bastaría con derivar otra clase de la original y modificar el comportamiento de los métodos necesarios. Claro, para esto la clase Coche debería estar bien construida. Teniendo los métodos :

```
public virtual void Acelerar(double cantidad)
{
// Aquí se le dice al motor que aumente las revoluciones pertinentes, y...
    Console.WriteLine("Accionando el mecanismo de aceleración por defecto");
    Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);
this.velocidad += cantidad;
}
public virtual void Girar(double cantidad)
{
    // Aquí iría el código para girar
    Console.WriteLine("Girando el coche {0} grados", cantidad);
}
public virtual void Frenar(double cantidad)
{
    // Aquí se le dice a los frenos que actúen, y...
    Console.WriteLine("Reduciendo la velocidad en {0} km/h", cantidad);
    this.velocidad -= cantidad;
}
```

virtual – override y el Polimorfismo por herencia :

Hemos añadido la palabra virtual en las declaraciones de los tres métodos. ¿Para qué? Para que las clases derivadas puedan sobrescribir el código de dichos métodos en caso de que alguna de ellas lo necesite porque haya cambiado el mecanismo.

Fíjate bien en cómo lo haría una clase que sobrescribe el método Acelerar porque utiliza un sistema distinto al de la clase Coche:

```
class CocheAceleradorAvanzado:Coche
{
    public CocheAceleradorAvanzado(string marca, string modelo, string color, string
                                   patente): base(marca, modelo, color, patente) {}
    public override void Acelerar(double cantidad)
    {
    // Aquí se escribe el nuevo mecanismo de aceleración
        Console.WriteLine("Accionando el mecanismo avanzado de aceleración");
        Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);
        velocidad += cantidad;
    }
}
```

Ya está. La clase base queda intacta, es decir, no hay que modificar absolutamente nada.

Conclusiones:

- La clase derivada únicamente sobrescribe aquello que no le sirve de la clase base, que es en este caso el método acelerar. Usando la palabra ***override*** en la declaración del método.
- El compilador siempre ejecuta el método sobrescrito si el objeto pertenece a la clase derivada que lo sobrescribe.

Veamos un ejemplo:

```
static void Main()
{
    CocheAceleradorAvanzado miCoche;
    miCoche = new CocheAceleradorAvanzado("Peugeot", "306", "Azul", "54668742635");
    miCoche.Acelerar(100);
}
```

En este caso, está muy claro. El objeto *miCoche* está declarado como un objeto de la clase *CocheAceleradorAvanzado*, de modo que al ejecutar el método acelerar se ejecutará sin problemas el método de la clase derivada. Por lo tanto, la salida por pantalla de este fragmento sería:

```
Accionando el mecanismo avanzado de aceleración
Incrementando la velocidad en 100 km/h
```

Sin embargo, este otro ejemplo puede ser más confuso:

```
static void Main()
{
    Coche miCoche;
    miCoche = new CocheAceleradorAvanzado("Peugeot", "306", "Azul", "54668742635");
    miCoche.Acelerar(100);

}
```

Aquí el objeto *miCoche* está declarado como un objeto de la clase *Coche* y, sin embargo, se instancia como objeto de la clase *CocheAceleradorAvanzado*.

¿Cuál de los dos métodos ejecutará ahora?

De nuevo ejecutará el método de la clase derivada, como en el caso anterior. Por más que se haya declarado el objeto *miCoche* como un objeto de la clase Coche.

Esto sucede porque el operador *override* extiende la implementación de la clase base *Coche*. Como el tipo instanciado es de la clase derivada *CocheAceleradorAvanzado*, se ejecuta el método de la clase derivada.

La salida por pantalla en este caso sería, por lo tanto, exactamente la misma que en el caso anterior.

Este tipo de polimorfismo se conoce como **"Polimorfismo por herencia o sobreescritura"** y, en resumen, ofrece la posibilidad de que varios objetos que comparten la misma interfaz, es decir, que están formados por los mismos miembros, se comporten de un modo distinto unos de otros.

Cuando usar el modificador *new* en lugar del *override*

El modificador *new* se utiliza cuando tengo la intención de hacer una implementación en una clase heredada que esconda la implementación de la clase base.

```
using System;


namespace OverrrideAndNew
{
    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
```

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        bc.Method2();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
        bcdc.Method2();
    }
    /* Output:
        Base - Method1
        Base - Method2
        Derived - Method1
        Derived - Method2
        Derived - Method1 //Because override makes to use the method from the
derived class
        Base - Method2  //Because Method2 implementation are different (due to
new usage) in Base and Derived
                        //Classes. bcdc reference is type of BaseClass, then
it access to its implementation.
    */
}
```

Clases base

Una declaración de clase puede especificar una clase base colocando después del nombre de clase y los parámetros de tipo dos puntos seguidos del nombre de la clase base. Omitir una especificación de la clase base es igual que derivarla del tipo *object*.

Accesibilidad

En C#, cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa que pueden tener acceso al miembro. Existen seis formas de accesibilidad posibles. Se resumen a continuación.

`public`: Acceso no limitado

```
public class Bicicleta
{
    public void Pedalear() { }
}
```

`protected`: Acceso limitado a esta clase o a las clases derivadas de esta clase

```
public class Bicicleta
{
    protected void Pedalear() { }
}

public class BicicletaMotorizada : Bicicleta
{
    public void EncenderMotor() { }
}
```

`Internal`: Acceso limitado al ensamblado actual (.exe, .dll, etc.)

`protected internal o internal protected`: Solo para miembros. Acceso limitado dentro del mismo assembly, Para las clases que extienden, el acceso se puede hacer dentro o fuera del ensablado. Os sea que es un `internal + protected`.

`private`: Acceso limitado a esta clase.

`private protected`: Solo para miembros. El acceso es limitado a los tipos derivados dentro del assembly que los contiene.

Si una clase no tiene modificador, su accesibilidad es: `internal`
Si un miembro no tiene modificador, su accesibilidad es: `private`

BIBLIOGRAFÍA RECOMENDADA:

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels
https://blogs.msdn.microsoft.com/mazhou/2017/10/05/c-7-series-part-5-private-protected/

Head First C# 3rd Edition Jennifer Greene Andrew Stellman