# Curso de Programación .NET nivel I

# Unidad 3

# “Programación de Objetos en .NET”

Contacto: consultas@elearning-total.com
Web: www.elearning-total.com

# Índice

## Contenido



# Introducción

En esta unidad vamos a introducir los conceptos básicos de la programación orientada a objectos utilizando C#, ya que es el lenguaje que hemos estado aprendiendo en la anterior unidad.

La programación orientada a objetos es una manera de programar con su conjunto de reglas y buenas prácticas. Si quisiéramos podríamos elegir otro tipo de programación: la programación estructurada, como hicimos en la unidad anterior; esta elección depende del nivel de complejidad de nuestra aplicación.

C# nos da muchas facilidades para programar en objetos para situaciones de mediana a alta complejidad.

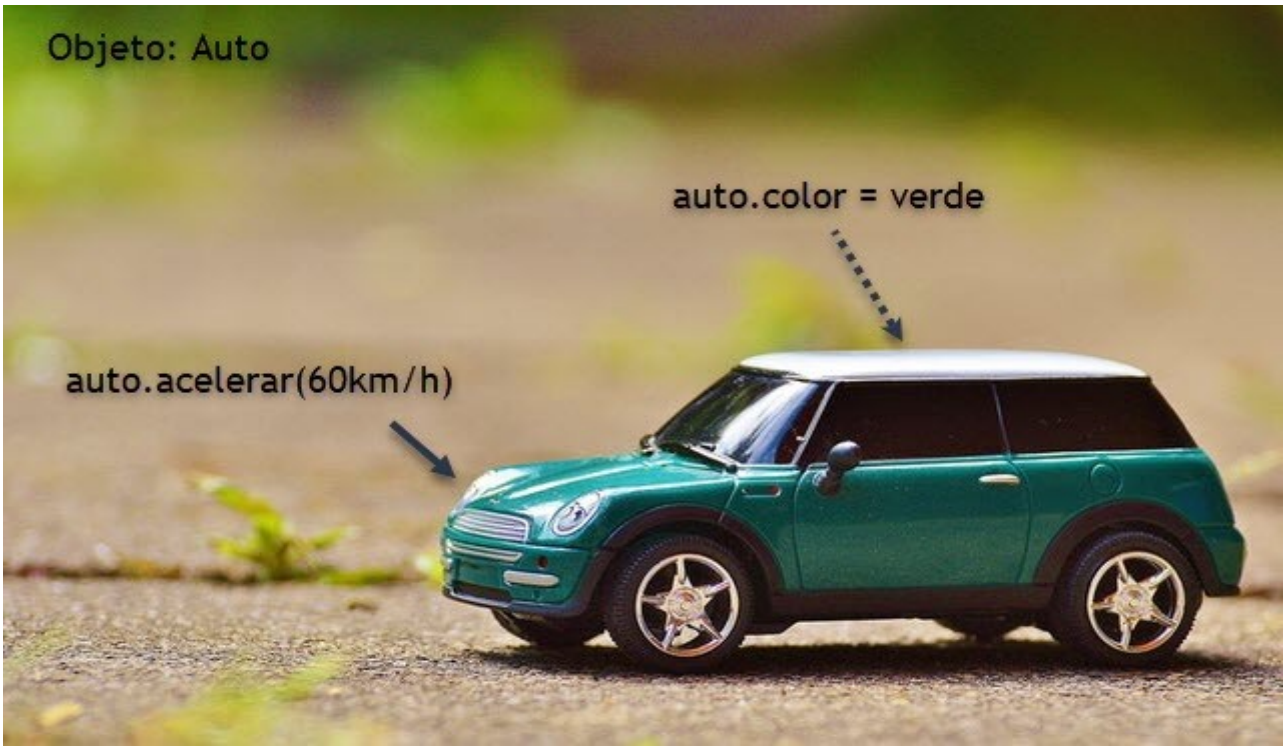
Por último y no menos importante, la Programación orientada a objetos es uno de los paradigmas más aceptados y utilizados en el mundo, por lo tanto, es indispensable para cualquier programador que quiera insertarse en el mercado laboral. En las entrevistas laborales siempre suelen preguntar sobre conceptos de objetos.

# La programación orientada a objetos

La programación orientada a objectos (OOP por sus siglas en inglés) es una manera de programar mucho más cercano a la vida real si lo comparamos con otros tipos de programación.

En OOP tenemos que pensar en términos de objetos juntos con sus propiedades y métodos. El ejemplo clásico es el del automóvil en el cual el coche es el elemento principal junto a un número de características como ser el color, la marca o el modelo; Además el coche puede realizar acciones, digamos acelerar y frenar.

Si queremos modelar el automóvil en OOP nos quedaría el Auto como el **objeto**, luego sus **propiedades** serian color, marca y modelo y por último los **métodos** serian las acciones del automóvil: acelerar y frenar.




Este objeto se podría utilizar en la vida real en un programa para una concesionaria o un taller de autos, en el que tenemos que gestionar una lista de coches para repararlos o venderos.

## Las Clases

Si el objeto es una estructura especifica con propiedades singulares, como por ejemplo: color verde o tipo 3 puertas, la clase es su abstracción o dicho de manera coloquial es el “molde”:

Un objeto siempre va a pertenecer a alguna clase ya que en ella tiene las declaraciones de sus propiedades y métodos, por ejemplo: La clase auto va a contener las propiedades color, tipo y marca y los métodos Acelerar() y Frenar(). Entonces tenemos dos componentes fundamentales en las clases:

- **Propiedades**:
  Son las características o atributos de los objetos. Para referirnos a las las propiedades tenemos que definir tipo y nombre; Podemos decir que las propiedades son las variables de un objeto.
- **Métodos**:
  Son las acciones que tengo a disposición en un objeto. Como en el caso de las propiedades los métodos tienen un tipo, nombre y también pueden tener argumentos; Podemos decir que los métodos son las funciones de los objetos.

```
class Auto
  {

        //PROPIEDADES
    public int puertas;
    public string color;
    private int velocidad;

        //METODOS

    public void Acelerar(int velocidadNueva)
    {
      this.velocidad = velocidadNueva;
    }


  }
```

Aquí arriba vemos como se declara una clase básica, fíjense que ya vimos estos conceptos en la unidad anterior la diferencia es que antes lo hacíamos todo dentro de la case Main. Lo nuevo que aparece acá es “private” y “public” que tienen que ver con la visibilidad, pero antes de llegar a ese punto tenemos que ver otros temas.

## Los Objetos

Los objetos son ejemplares de una clase, por lo tanto al crear un objeto a través de una clase vamos a definir que valores tienen sus propiedades, como también vamos a tener disponible el uso de las funciones de esa **instancia** de la clase.

**Instanciar** es crear un objeto a partir de su molde o mejor dicho de su clase y se hace de la siguiente manera:

```
Auto miAuto = new Auto();
```

En esta instrucción en la parte de la izquierda lo que estoy haciendo es declarar una variable de tipo clase Auto que voy a usar para alojar un objeto de ese tipo.
En la segunda parte estoy instanciando un objeto con la palabra new, lo que hacemos es crear una instancia de la clase que hemos escrito seguido de paréntesis. Dentro de los paréntesis podríamos colocar parámetros con los que inicializar el objeto de la clase Auto.

Esta variable de tipo clase me permite referirme a mi objeto para acceder a sus propiedades o métodos.

## Estado de un objeto

Al valor concreto de una propiedad de un objeto, que lo hace único, es lo que se denomina estado.

```
miAuto.color = "verde";
```

El objeto en este caso es miAuto y luego del mismo escribimos un punto para acceder a sus propiedades, ya sea para modificarlas o para leerlas.

## Mensajes de un objeto

El mensaje a un objeto no es más que una llamada a una de sus funciones. Para llamar a un objeto lo que debemos hacer es escribir la variable que lo contiene y seguido de un punto escribir la función que deseamos invocar.

```
miAuto.Acelerar(80);
```

En las unidades anteriores, si recordamos bien, ya usamos muchos de estos conceptos.
Por ejemplo: la función de mostrar texto en pantalla:

```
Console.WriteLine("Hola Mundo");
```

Sigue la misma estructura que ya vimos, al objeto Consola le paso el mensaje Writeline con el parámetro “hola Mundo”.

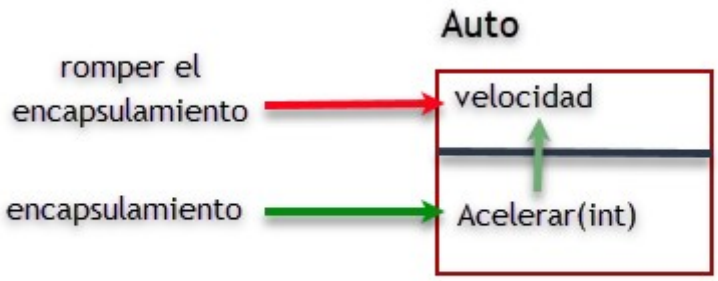
(Aclaración: en realidad Consola en este caso es una clase ya que no la estoy instanciando por lo tanto es posible llamar a métodos directos de la clase, pero no quiero confundirlos por ahora simplemente quería que vean que estuvimos viendo estos conceptos a lo largo de las clases, si se quedaron con eso por ahora es más que suficiente)

## Encapsulamiento

Según el paradigma de objetos, la lógica se divide en objetos los cuales van a tener cierto rol o responsabilidad en nuestro modelo, el cual debería seguir cierta lógica valga la redundancia.

Un auto debería ser el responsable de modificar su velocidad interna y no el conductor. El conductor lo que debería hacer es presionar el pedal o llamar al método Acelerar(60). El auto debe entender ese mensaje y modificar su velocidad, ya que es su responsabilidad.

El empaquetamiento de las variables o propiedades se lo llama encapsulamiento y sirve para esconder detalles de la implementación de las funciones.




Esto me permite cambiar esos detalles de implementación en el futuro sin afectar otras partes del programa.

Veamos el caso de un objeto auto con encapsulamiento:

```
public class Auto
    {
   public int velocidad;

   public void Acelerar(int velocidadNueva)
         {
      this.velocidad = velocidadNueva;
         }
    }

class  Main()
    {
    miAuto = new Auto()    //Sin respetar el encapsulamiento
     miAuto.velocidad = 80;
   }
```

En el ejemplo de arriba la propiedad velocidad no esta encapsulada ya que tiene la palabra clave `public`. La palabra clave para hacer que solo se pueda acceder a esa variable desde las funciones propias del objeto es `private.`

```
public class Auto
    {
   private int velocidad;

   public void Acelerar(int velocidadNueva)
         {
      this.velocidad = velocidadNueva;
         }
    }

class  Main()
    {
    miAuto = new Auto()    //Ya no puedo hacer esto:
     miAuto.velocidad = 80;
      //Tengo que respetar el encapsulamiento:
     miAuto.Acelerar(80);
      }
```

Como ven puedo aplicar `private` o `public` tanto a propiedades como métodos o clases.

```
    public void Acelerar(int velocidadNueva)
```

Esta manera de programar pensando en objetos lógicos, responsabilidades y encapsulamiento es clave en este paradigma y como es de esperar hay un grado de subjetividad en el mismo, ya que todos tenemos una

visión de la realidad distinta y por lo tanto podríamos modelar de distinta manera el mismo problema. Lo importante es ser consistente en las soluciones.

## Constructores

Recordemos cuando creábamos objetos a partir de una clase usando la expresión new:

```
Auto miAuto = new Auto();
```

En ese momento hicimos uso de un constructor que viene por defecto al crear una clase. El constructor es un método especial de una clase que tiene el fin de inicializar valores de los objetos al crear los objetos.

Por ejemplo:

```
Auto miAuto = new Auto("verde", 2007,  3 , "cooper");
```

En este ejemplo queremos crear un objeto y que ya venga con esos valores de color, año, cantidad de puertas y modelo, sin la necesidad de estar accediendo a las variables una por una en las lineas subsiguientes y así evitar repetir código.

Para poder hacer esta llamada necesitamos declarar el método constructor de la siguiente manera:

```
class Auto
      {
    private string color;
    private int año;
    private int puertas;
    private string modelo;
    private int velocidad;

    public Auto(string color, int año, int puertas, string modelo)
           {
      this.color = color;
      this.año = año;
      this.puertas = puertas;
      this.modelo = modelo;
           }
     }
```

El orden de ejecución es el siguiente

1. Se llama al constructor con la expresión new.
2. Se crea el objeto a partir de la clase especificada en 1.
3. Si tiene parámetros, se le pasan los parámetros al método constructor.
4. El método constructor toma los parámetros y se los asigna al objeto creado (this).

Algunas cosas a notar:

- El método constructor lleva el mismo Nombre que la clase,
- El constructor devuelve un objeto del tipo de la clase, en este caso Auto, por lo tanto no se le define ningún tipo en la declaración de la función.
- this apunta al objeto creado.

Ahora ustedes dirán que les mentí y que en realidad lo que hice fue trasladar la asignación de propiedades a un método y no gane mucho; Y pueden que tengan razón si solo quisiéramos crear un solo Auto, pero que sucede si queremos crear dos Autos:

```
Auto miAuto = new Auto();
miAuto.color = "verde";
miAuto.año = 2007;
miAuto.puertas = 3;
miAuto.modelo = "cooper";
Auto miAuto2 = new Auto();
miAuto2.color = "negro";
miAuto2.año = 2009;
miAuto2.puertas = 5;
miAuto2.modelo = "ka";
```

Bueno se ve un poco desordenado y además estoy rompiendo el encapsulamiento ya que estoy accediendo a variables privadas de los objetos.
Puedo reemplazar esas 8 lineas por dos métodos constructores:

```
Auto miAuto = new Auto("verde", 2007,  3 , "cooper");
Auto miAuto = new Auto("negro", 2009,  5 , "ka");
```

No solo termino ahorrando lineas de código sino que me permite estandarizar el ingreso de datos y eso evita que tenga un error por omisión de alguna propiedad.

## Getters and Setters

Como dijimos antes en la programación orientada a Objetos queremos evitar romper el encapsulamiento o bien, evitar acceder a propiedades que son privadas. Vimos que una manera de evitar esto es por medio de los constructores, aunque eso solo me resuelve el problema al iniciar el objeto.
Cuando queremos modificar una propiedad después de un tiempo de creado el objeto utilizamos métodos Getters and Setters y lo que hacen es darnos una manera de leer y escribir propiedades llamando a métodos (mensajes) y así evitamos romper el encapsulamiento.

Ejemplo:

```
private int color;

public void SetColor(string color)
        {
        this.color = color;
        }

public string GetColor()
        {
        return color;
        }
```

SetColor es el Setter y permite asignar un valor a la propiedad privada color y GetColor nos desvuelve el valor de la misma, hay que crear Setters y Getters por cada propiedad privada del objeto que queremos dar acceso. Los Getter y los Setter son muy convenientes si queremos hacer algún control o transformación al ingresar un dato, si esto no es necesario podemos hacer uso del tipo Property de Visual Studio que nos implementa la propiedad privada y los Getter y Setter todo en una sola línea:

```
public int MyProperty { get; set; }
```

## Resumen

Hasta acá vimos que las clases tiene definiciones propiedades y métodos. Los objetos se crean a partir de las clases y es ahí donde esas propiedades y métodos cobran vida.

Puedo crear la cantidad de objetos que desee desde la misma clase y cada uno puede tener distintas propiedades.

En la próxima unidad vamos a ver como las clases pueden interactuar entre sí tanto de manera jerárquica o agregada, lo que nos permite modelar escenarios más interesantes en los cuales un objeto puede ser parte de otro o heredar características y así resolver problemas complejos de manera simple.