Módulo 2

Programación Orientada a Objetos en .NET Core (C#)

1. Introducción a la Programación Orientada a Objetos

Curso de Programación en .NET

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial
Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro
Ayuntamiento de Málaga
pdø de contenidos digitales

Index

Concepto de Orientación a Objetos

Hasta más o menos mediados de los años 80, el paradigma de programación que se utilizaba era la programación estructurada. Básicamente se dividía el código en funciones y módulos (archivos), y teníamos un archivo o módulo principal donde se ejecutaba el programa. Desde este programa se utilizaban funciones definidas en ese u otros módulos para hacer algún tipo de cálculo, etc.

A partir de ese momento, el paradigma estructurado evolucionó al paradigma orientado a objetos. Un objeto es una entidad que encapsula un estado (propiedades o variables de objeto) y un comportamiento (métodos o funciones del objeto). Además, estas propiedades y métodos no tienen por qué ser accesibles desde fuera del mismo. Algunos métodos se utilizan para cálculos internos del propio objeto, y existen propiedades cuya modificación no queremos que se pueda realizar desde cualquier sitio, sino que es mejor controlar desde algún método del objeto.

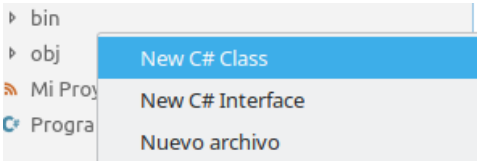
En un lenguaje orientado a objetos, todo se estructura en clases y objetos, desde el propio programa principal hasta las clases que representan los datos con los que vayamos a trabajar (productos, clientes, personajes de un juego, etc.).

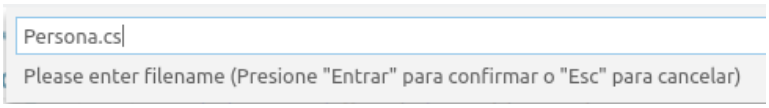
En resumen, la programación orientada a objetos nos permite crear aplicaciones que son **flexibles** (en cualquier momento se puede sustituir una clase por otra, o reutilizarla en otro programa), más cercanas al lenguaje **natural** (cada clase representa un concepto dentro del programa global), **seguras** (gracias a la encapsulación de propiedades y métodos podemos decidir qué es público o privado), y fáciles de **probar** (cada clase y método tiene un objetivo muy concreto → separación de responsabilidades).

Clases

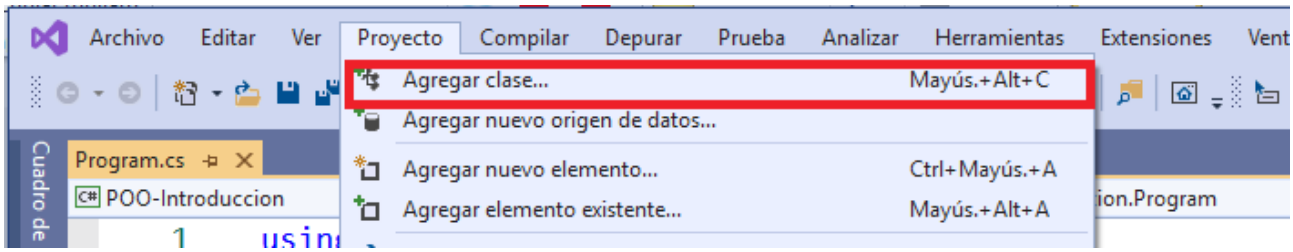
Una clase no es lo mismo que un objeto, aunque ambos conceptos tienen una fuerte relación. Una clase contiene la definición de las propiedades y métodos que los objetos derivados de dicha clase tendrán. Aunque podemos crear varios objetos a partir de una misma clase, estos serán diferentes ya que los valores de sus propiedades no serán iguales normalmente. Sin, embargo, todos tendrán la misma funcionalidad (métodos).

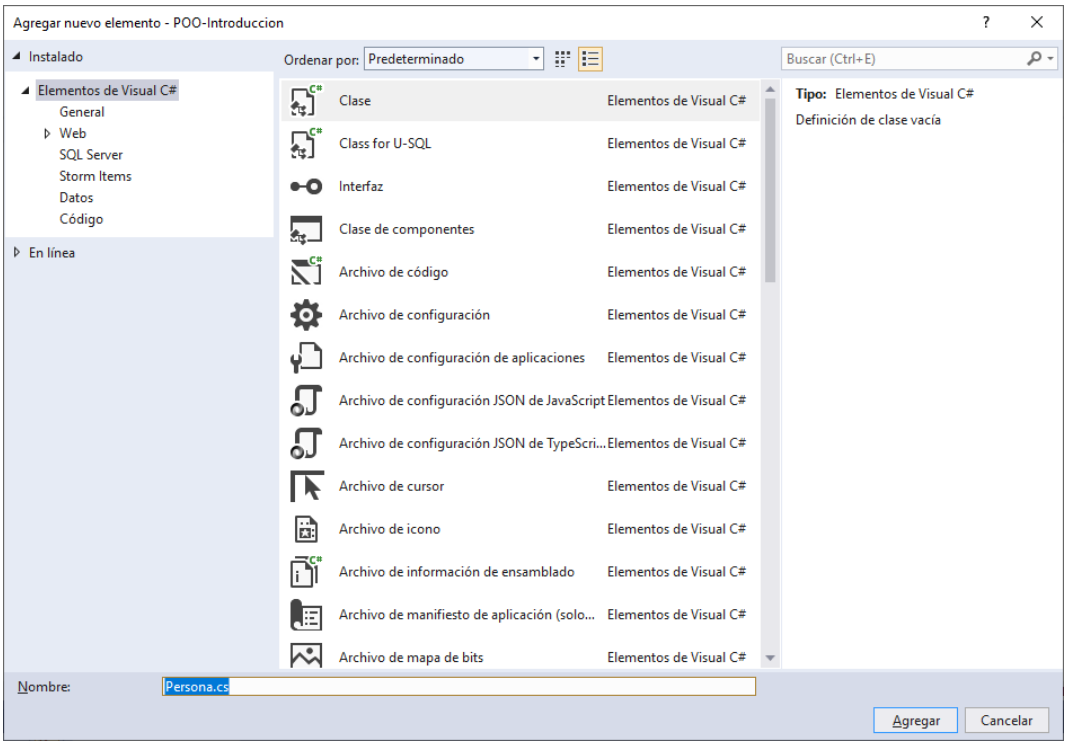
Para crear una clase, basta con crear primero un archivo que se llame igual que la clase con la extensión **.cs** (CSharp). Si hemos instalado las extensiones de C# en Visual Studio Code, podremos hacer esto directamente con el botón derecho del ratón donde se encuentran los archivos del proyecto.





En Visual Studio 2019 tenemos que seleccionar la opción “Agregar Clase” del menú “Proyecto”





Esto nos creará el archivo con el siguiente código:

```
namespace Mi_Proyecto
{
    class Persona
    {
    }
}
```

Como podemos ver, primero define un espacio de nombres (namespace) “Mi_Proyecto” y dentro crea la clase Persona. Lo normal es que todas las clases de un mismo proyecto estén en el mismo espacio de nombres. Aunque se puede omitir crear un espacio de nombres, es recomendable para que los nombres de la clases no colisionen con las clases que podamos utilizar de alguna librería (ya aprenderemos el concepto de librería más adelante).

Dentro de la clase vamos a crear una propiedad llamada **Nombre** de tipo string. De esta manera vamos a ver mejor como crear objetos diferentes a partir de una clase. En breve explicaremos más sobre las propiedades de una clase.

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

Para **instanciar** un objeto a partir de la clase definida, usamos la palabra reservada **new** delante del nombre de la clase. Esto nos devolverá un objeto de dicha clase. A partir de ahí podemos empezar a modificar o leer sus propiedades. Esto lo haremos en **Program.cs** dentro del Main.

```
static void Main(string[] args)
{
    Persona p1 = new Persona();
    p1.Nombre = "Marta";
    Persona p2 = new Persona();
    p2.Nombre = "Joaquín";
    Console.WriteLine($"He creado las personas {p1.Nombre} y {p2.Nombre}");
}
```

Propiedades

Las propiedades de un objeto son como variables ligadas a la existencia del mismo. Definen las características del objeto. Estas propiedades se definen en una clase como si fueran variables pero generalmente con la palabra clave **public** delante (esto es un modificador de ámbito, en breve veremos más sobre esto).

Además, debemos indicar si la propiedad va a poder leerse fuera de los métodos del objeto (**get**) o si podrá modificarse (**set**). A diferencia de una variable, en C# los nombres de las propiedades suelen empezar por mayúscula (esto no es así en otros lenguajes como Java, por ejemplo).

```
public class Producto
{
    public string Nombre {get; set;}
    public double Precio {get; set;}
}


Producto p = new Producto();
p.Nombre = "Armario";
p.Precio = 135.5;
```

A veces necesitamos controlar los valores que se asignan a una propiedad (**set**), o procesar la propiedad antes de devolverla (**get**). En ese caso se suele declarar una propiedad privada llamada **campo de respaldo** (su nombre normalmente empezará por '_' o por minúscula) que es la que almacenará el valor, y otra propiedad pública que implementará las propiedades get y set y trabajará con la propiedad privada.

Lo que hemos hecho previamente se llaman **propiedades implementadas automáticamente** y es una versión resumida de lo siguiente (observa que creamos una propiedad privada asociada en minúsculas). El valor que recibe el set se representa con la palabra **value**:

```
public class Producto
{
    private string nombre;
    private double precio;

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }
    public double Precio
    {
        get { return precio; }
        set { precio = value; }
    }
}
```

De esta manera podemos crear propiedades e implementar una lógica de programación opcionalmente si lo necesitamos:

```
public class Cuadrado
{
    private double lado;

    public double Lado
    {
        get { return lado; }
        set
        {
            if (value > 0.0)
                lado = value;
        }
    }
}
```

```
Cuadrado c = new Cuadrado();
c.Lado = 4.5;
Console.WriteLine(c.Lado); // 4,5
c.Lado = -23; // Negativo, no se asigna
Console.WriteLine(c.Lado); // 4,5
```

De hecho se pueden crear propiedades, normalmente de solo lectura, que te devuelvan un valor calculado, aunque esto también se puede hacer creando un método, sin embargo es muy normal hacerlo usando propiedades calculadas en C#.

```
public class Cuadrado
{
    ...

    public double Area
    {
        get { return lado * lado; }
    }
}

Cuadrado c = new Cuadrado();
c.Lado = 4;
Console.WriteLine(c.Area); //16
```

En otros lenguajes como Java por ejemplo, se utilizan métodos que normalmente se nombran con el prefijo set o get y el nombre de la propiedad. También se puede hacer lo mismo en C# aunque no es habitual hacerlo en este lenguaje por lo que se suele evitar esta práctica.

```
public class Cuadrado
{
    private double lado;

    public double getLado()
    {
        return lado;
    }

    public void setLado(double lado)
    {
        if (lado > 0.0)
            this.lado = lado;
    }
}

Cuadrado c = new Cuadrado();
c.setLado(4.5);
Console.WriteLine(c.getLado()); // 4,5
c.setLado(-23); // Negativo, no se asigna
Console.WriteLine(c.getLado()); // 4,5
```

Constructor

El constructor es una función con el mismo nombre que la clase, y sirve para inicializar sus propiedades. Cuando se crea un objeto a partir de una clase con **new**

nombre_Clase(), se invoca automáticamente su constructor. Es decir, se ejecuta solo una vez en la creación de cada objeto.

Cuando una clase no tiene ningún constructor definido, aún así se pueden crear objetos utilizando el constructor por defecto. Es equivalente a definir un constructor vacío, sin parámetros y sin implementación.

```
public class Cuadrado
{
    // Propiedades

    public Cuadrado() {}

    // Métodos
}
```

Los constructores que reciben algún tipo de parámetro se denominan **constructores parametrizados**. Normalmente recibirán tantos parámetros como propiedades vayan a inicializar (a no ser que inicialicen con valores por defecto o calculados a partir de otros parámetros).

```
public class Cuadrado
{
    // Propiedades

    public Cuadrado(double lado) {
        Lado = lado;
    }

    // Métodos
}
```

Una clase puede tener varios constructores. Siempre y cuando haya diferencia en el tipo u orden de los parámetros. De esta manera cuando creamos un objeto, el programa sabrá a qué constructor se debe llamar en función de los parámetros recibidos. A esto se le denomina **sobrecarga de constructores**.

```
public class Cuadrado
{
    …

    public Cuadrado()
    {
        Lado = 1;
    }

    public Cuadrado(double lado)
    {
        Lado = lado;
    }

    public Cuadrado(int lado)
    {
        Lado = lado;
    }
}
```

```
Cuadrado c = new Cuadrado();    // al crearlo sin parámetros va al constructor
Console.WriteLine(c.Lado); //1 // sin parámetros cuyo Lado =1

Cuadrado c2 = new Cuadrado(2); // al crearlo con un parámetro entero
Console.WriteLine(c2.Lado); //2 // va al constructor con parámetro entero
```

Métodos

Además de las propiedades y constructores, hay otro elemento principal que define una clase: los métodos. Un método es una función asociada a una clase y que agrega un comportamiento a los objetos creados a partir de la misma.

Para llamar a un método basta con poner el objeto (la variable que lo contiene por ejemplo) seguido de un punto y el nombre del método (con los correspondientes parámetros).

```
public class Producto
{
    public string Nombre { get; set; }
    public double Precio { get; set; }

    public Producto(string nombre, double precio) // constructor parametrizado
    {
        Nombre = nombre;
        Precio = precio;
    }

    public double getPrecioImpuesto(double impuesto)  // método
    {
        return Precio * (1 + impuesto);
    }
}

Producto p = new Producto("Silla", 45.90);
double precioFinal = p.getPrecioImpuesto(0.21);
// La silla cuesta 55,54
Console.WriteLine($"La silla cuesta {precioFinal:F2}");
```

Un método puede recibir parámetros de cualquier tipo y devolver un valor. También puede acceder a cualquier propiedad del objeto actual. Es importante tener en cuenta de que en función del objeto sobre el que llamemos al método, los valores de las propiedades serán diferentes, es decir, serán las del objeto actual. Todo lo que hemos aprendido referente a las funciones previamente es aplicable a los métodos.

Los métodos también admiten **sobrecarga**, es decir, podemos tener varios métodos con el mismo nombre siempre y cuando los parámetros (número y/o tipo) sean diferentes. Se llamará al método que concuerde con los parámetros recibidos.

```
public class Producto
{
    ...
```

```
    // Recibimos el impuesto a aplicar
    public double getPrecioImpuesto(double impuesto)
    {
        return Precio * (1 + impuesto);
    }

    // No recibe nada, aplicamos el impuesto por defecto (21%)
    public double getPrecioImpuesto()
    {
        return Precio * 1.21;
    }

    ...
}
```

Concepto de copia y referencia (parámetros y asignaciones)

Cuando asignamos una variable que contiene un objeto a otra variable (o a una posición de un array, por ejemplo), no se genera automáticamente una copia del objeto como ocurre con tipos primitivos de datos (int, double, char, string,...). En su lugar ambas variables apuntarán al mismo objeto, por lo que si modificamos algo desde una variable, tendrá efecto al acceder desde la otra.

```
Producto prod = new Producto("Silla", 45.90);
Producto otroProd = prod;
otroProd.Precio = 100;
Console.WriteLine(prod.Precio); // 100
```

Cuando pasamos un parámetro a un método, pasa exactamente lo mismo, no se genera una copia del objeto, sino que se pasa una referencia al objeto original. Esto quiere decir que si modificamos las propiedades de un objeto que hemos recibido por parámetro dentro del método, estas modificaciones son permanentes y siguen vigentes una vez terminado el método. Se debe tener siempre en cuenta y andar con mucho cuidado por los posibles efectos colaterales que esto puede ocasionar.

```
class Program
{
    public static void ModificaProducto(Producto p)
    {
        p.Precio = 200;
    }

    static void Main(string[] args)
    {
        Producto prod = new Producto("Silla", 45.90);
        ModificaProducto(prod);
        Console.WriteLine(prod.Precio); // 200

    }
}
```

En el caso de los arrays pasa lo mismo que con los objetos.

```
class Program
{
```

```
    public static void ModificaArray(int[] a)
    {
        for(int i = 0; i < a.Length; i++)
        {
            a[i] *= 2;
        }
    }

    static void Main(string[] args)
    {
        int[] nums = {1, 5, 15, 25, 50};
        ModificaArray(nums);
        Console.WriteLine(String.Join(", ", nums)); // 2, 10, 30, 50, 100

    }
}
```

Una opción para evitar esto sería generar una copia del objeto usando un constructor de copia (recibe un objeto de esa clase y genera otro objeto con los mismos valores en las propiedades).

```
public class Producto
{
    public string Nombre { get; set; }
    public double Precio { get; set; }

    public Producto(string nombre, double precio)
    {
        Nombre = nombre;
        Precio = precio;
    }

    public Producto(Producto p) // Constructor de copia
    {
        Nombre = p.Nombre;
        Precio = p.Precio;
    }
    ...
}
```

```
Producto silla = new Producto("Silla", 50);
Producto copiaSilla = new Producto(silla); // Copiamos objeto

copiaSilla.Precio = 100;
Console.WriteLine(silla.Precio); // 50
```

Para generar una copia de un array o de un objeto tenemos el método Clone. Esto creará un nuevo array del mismo tamaño copiando los elementos del array original. Sin embargo, almacena los valores como objetos genéricos (tipo Object) por lo que debemos indicarle el tipo usando un **casting**.

```
int[] a = {12, 23, 42, 15};
int[] b = (int[])a.Clone();
b[0] = 99;
Console.WriteLine(String.Join(",", a)); // 12,23,42,15
Console.WriteLine(String.Join(",", b)); // 99,23,42,15
```

Sin embargo, hay que tener en cuenta que si un array contiene objetos (o clonamos un objeto que a su vez también contiene objetos), la copia generada tendrá la misma referencia a esos objetos internos. Una opción sería crear una copia en profundidad generando copias de esos objetos internos en la copia del array.

```
Producto[] prods = {
    new Producto("Manzana", 0.56),
    new Producto("Pera", 0.64)
};
Producto[] prods2 = new Producto[prods.Length]; // Nuevo array mismas posiciones
for(int i = 0; i < prods.Length; i++)
{
    prods2[i] = new Producto(prods[i]); // Copia de objeto

}
prods2[0].Nombre = "Puerro";
Console.WriteLine(prods[0].Nombre); // Manzana
```

Ámbito de las propiedades y métodos

Tanto las propiedades de la clase como sus métodos suelen empezar por una palabra que generalmente es **public** o **private**. Esto es lo que se conoce como modificadores de acceso. Si no se indica nada, el valor por defecto es **internal**, que es similar a public, con la excepción de que si en lugar de un programa estamos haciendo una librería, no podrían acceder las clases externas a esa librería. Más información aquí.

Los tipos de modificadores más usados son:

public → Un método o propiedad público es aquel puede ser utilizado desde cualquier parte, es decir, el código de una clase diferente podría llamar al método o acceder para leer o modificar una propiedad pública.

private → Cuando es privado, solo los métodos de la misma clase pueden llamar a ese método o acceder a esa propiedad. Esto se utiliza para métodos auxiliares a otros métodos públicos que no tiene sentido que se utilicen fuera de la clase actual. En el caso de las propiedades, ya hemos visto que muchas veces hay una propiedad privada y otra pública (que empieza en mayúscula) con los getters y setters para controlar el acceso a la privada.

protected → Es similar a private, con la diferencia de que si creamos una clase derivada (aún no hemos visto el concepto de herencia), también puede utilizarlo.

Métodos y propiedades de clase (static)

El modificador static se utiliza en las propiedades o métodos de una clase para indicar que no están vinculados a ningún objeto de la misma, sino a la propia clase. A estas propiedades o métodos estáticos los podemos llamar desde los métodos

normales (de instancia) de la clase, o desde fuera usando directamente el nombre de la clase.

```
public class Producto
{
    public static int TotalProductos = 0;
    ...

    public static void IncrementaProductos()
    {
        TotalProductos++;
    }
}

Producto p = new Producto("Silla", 50);
Producto.IncrementaProductos();
Console.WriteLine(Producto.TotalProductos); // 1
```

El valor null

El valor **null** implica una referencia que no apunta a ningún objeto. Es decir, es la manera de indicar que una variable, o una posición de un array está vacía. Cuando una variable apunta a null no podemos llamar a ningún método o acceder a ninguna propiedad o se lanzará un error del tipo **NullReferenceException**.

```
Producto p = null;
string nombre = p.Nombre; // ERROR
```

Debemos tener cuidado que no recibamos valores nulos en los parámetros de una función, se aconseja comprobarlo siempre que esto pueda ocurrir para evitar errores no deseados.

```
public static int CalculaMedia(int[] a)
{
    if(a == null)
 // No hay ningún array (nulo)
    {
        return 0;

    }


    // Calculamos la media aquí
}
```

El objeto this

La palabra reservada **this** no se puede utilizar en métodos estáticos ya que hace referencia al objeto actual sobre el cual estamos llamando al método. Cuando trabajamos con una variable o un parámetro que contiene un objeto, utilizamos ese nombre para acceder al objeto (propiedades, métodos), pero dentro de los métodos del propio objeto, usando this se accede a la referencia del mismo.

Sin embargo, la palabra this está implicíta por lo que podemos acceder a las propiedades y métodos del objeto actual sin poner this (ni nada) delante. En otros lenguajes de programación como JavaScript o PHP, es obligatorio poner siempre la palabra this.

```
public class Cuadrado
{
    private double lado;

    public double Area
    {
        get { return lado * lado; }
    }
}
```

```
public class Cuadrado
{
    private double lado;

    public double Area
    {
        get { return this.lado * this.lado; }
    }
}
```

En el caso de que haya ambigüedad. Por ejemplo, recibimos un parámetro con el mismo nombre que una propiedad del objeto, debemos usar **this** obligatoriamente para diferenciar el parámetro de la propiedad.

```
public class Cuadrado
{
    private double lado;

    public Cuadrado(double lado)
    {
         this.lado = lado;    // "this.lado" es la propiedad y "lado" el
parámetro
    }
    ...
}
```

También en el caso de querer pasar por parámetro (o guardar en un array por ejemplo) el objeto actual, debemos acudir a la palabra this.

```
public class Producto
{
    public static int TotalProductos = 0;
    public string Nombre { get; set; }
    public double Precio { get; set; }
    private Categoria cat;
    ...
    public CambiaCategoria (Categoria nueva)
    {
        cat.EliminaProducto(this);
        nueva.AddProducto(this);
        cat = nueva;
    }
}
```