# **INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS (POO)**

## **Diferencias entre Programación Estructurada y POO**

### 

### **Programación Estructurada**

La programación estructurada nació como solución a los problemas que presentaba la programación no estructurada, la cual se empleó durante mucho tiempo antes de la invención de la programación estructurada.

Un programa no estructurado es un programa procedimental: las instrucciones se ejecutan en el mismo orden en que han sido escritas. Sin embargo, este tipo de programación emplea la instrucción "goto". Una instrucción "goto" permite pasar el control a cualquier otra parte del programa, es decir “hacer saltos a otra sección del código”. Cuando se ejecuta una instrucción "goto" la secuencia de ejecución del programa continúa a partir de la instrucción indicada en el salto.

### **Programación orientada a objetos**

La programación orientada a objetos o POO (OOP – Object Oriented Programming según sus siglas en inglés), es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas que son sus pilares conceptuales, incluyendo herencia, abstracción, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existen variedad de lenguajes de programación que soportan la orientación a objetos.

Los objetos son entidades que tienen un determinado estado formado por sus **atributos**, tienen **comportamiento** a través de sus métodos, y poseen identidad:

* El estado está compuesto de datos, será uno o varios atributos a los que se habrán asignado valores concretos.
* El comportamiento está definido por los métodos o mensajes a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.
* La identidad es una propiedad de un objeto que lo diferencia del resto, dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un objeto se construye a partir de la clase que lo define. La clase es la plantilla del objeto, un molde.

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

**Beneficios de la Programación Orientada a Objetos**

La programación orientada a objetos aporta a los desarrolladores los medios de enfrentarse a sus nuevos retos, con:

* Una organización modular muy cercana a la realidad, pues un sistema se piensan en los “objetos que interactúan entre sí”.
* Procesos de creación, puesta a punto y mantenimiento de los componentes, más sencillos y rápidos.
* La reutilización y evolución de los componentes existentes o de aquellos que provienen de proveedores de software de terceros.
* Una integración sencilla para su funcionamiento en entornos gráficos.
* Una lógica de codificación compatible con las aplicaciones distribuidas, que reparten sus contenidos en varias máquinas.
* Un desacoplamiento de la aplicación, que permite un trabajo en equipo más eficaz y productivo. Esto significa que no existe código repetido y solapado (que hace lo mismo), en distintas partes de un sistema.

Y los principios básicos de la POO son:

* **Abstracción**: denota las características esenciales de un objeto del mundo real, objeto tangible o intangible, donde se capturan sus comportamientos. El comportamiento correcto en POO consiste en establecer relaciones entre objetos lo más abstractas posible. Por ejemplo, un origen de datos puede provenir de una entrada por teclado, del contenido de un archivo, de una conexión de red, etc. Si determinada funcionalidad de un objeto recibe como argumento una sucesión de datos, será interesante abstraerse de su origen, considerándola como un flujo "genérico de datos" (*stream*), sin pensar si viene del teclado o de un archivo, etc.
* **Encapsulamiento**: significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. El encapsulamiento consiste en crear un tipo de **caja negra** que contiene internamente el comportamiento del objeto, y externamente un conjunto de comandos que van a permitir manipularlo desde el exterior, esto implica que el usuario del objeto jamás podrá acceder internamente a sus datos, sólo lo hará desde los comandos.
* **Principio de ocultación**: cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar unos con otros.
* **Herencia**: Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. Por ejemplo, crear una clase de base, llamada *Empleado*, que contiene la información común a todos los empleados de una empresa. Después crear una clase llamada *Ejecutivo*, que heredará de la clase base *Empleado*. *Ejecutivo* heredará los miembros (datos y funcionalidad) de *Empleado* y agregará a esta lista de miembros los aspectos específicos del tipo ”ejecutivo”. En este caso, se dice que *Ejecutivo* extiende *Empleado* o que *Ejecutivo* hereda de *Empleado*. *Ejecutivo* es la clase derivaday *Empleado* es la clase base*.*
* **Polimorfismo**: permite tener comportamientos similares, asociados a objetos distintos. El polimorfismo de los objetos está muy relacionado con la herencia. La raíz etimológica de la palabra conduce a pensar de manera natural que el objeto puede adoptar varias formas. Para entender en qué medida esto es posible, imaginemos que tenemos un objeto *Ingeniero* que hereda de la clase *Ejecutivo* y, por tanto, un objeto *Ingeniero* es un tipo de *Ejecutivo*. Gracias al polimorfismo, en cualquier sitio donde se espere un objeto *Ejecutivo* se podrá utilizar un objeto *Ingeniero*.

# **LOS TIPOS DE C#**

**En C#, todo está tipado**

El término genérico "**tipo**" agrupa las clases, estructuras, interfaces, enumeraciones y delegados. Estos cinco tipos se describen en la CTS (*Common Type System*) para que los compiladores de diferentes lenguajes puedan generar código explotable por la CLR (*Common Language Runtime*). Un programa utiliza diferentes tipos y un ensamblado puede implementar varios tipos. A continuación se muestran las definiciones resumidas de los diferentes tipos propuestos por C#:

* El tipo "**Clase**" es la implementación C# de la clase del paradigma de objetos. Evidentemente es el tipo más utilizado en las aplicaciones.
* El tipo "**Estructura**" es una herencia del lenguaje C. Antes de la democratización de la programación orientada a objetos, las estructuras eran el medio más común para los desarrolladores para construir sus propios tipos. De momento, nos quedamos con que las estructuras de C# son muy cercanas a las clases y que, cuando se utilizan sabiamente, permiten mejorar el rendimiento de una aplicación. Las estructuras no existen en Java.
* El tipo "**Interfaz**" es muy utilizado en el framework .NET y contribuye a la comunicación entre las clases. Por el momento nos quedamos con que una interfaz es una clase sin código que formaliza un lote de métodos obligatorios para la clase que la implementa.
* El tipo "**Enumeración**" permite la definición de listas clave-valor y la creación de datos cuyos contenidos se limitarán a estas claves. Por ejemplo, se puede crear un tipo *Día* que puede contener de *lunes* a *domingo*. En C#, este tipo aporta un lote de métodos que permite gestionar esta lista por programación.
* El tipo "**Delegate**" (Delegado) encapsula la noción de puntero de función de C/C++, origen de buena parte de los problemas, empezando por asignarle un tipado fuerte. De hecho, el puntero de función "convencional" no es otra cosa que una dirección de memoria sin ninguna otra precisión sobre la firma, de modo que la aplicación se detiene con un error cuando los argumentos que se pasan no se corresponden con los argumentos esperados. Por este motivo el tipo *delegate* de C# se va a definir de manera precisa con la firma del método que se le asocia. Seguidamente, la instancia de tipo *delegate*, generalmente creada dentro de una clase que establece la comunicación con otras, gestiona una lista "de abonados" a través de una sintaxis desconcertante por su simplicidad. De hecho, basta con utilizar el operador += del *delegate* para registrarse como abonado a la lista de difusión y -= para eliminarse de la misma. Los *delegate* se utilizan mucho en C#; los encontraremos habitualmente en las interfaces gráficas para que los componentes puedan notificar a la aplicación sus cambios de estado. Este es un tema avanzado que no abordaremos en este curso.

**"Todo el mundo hereda de System.Object"**

El tipo *System.Object* es la base directa o indirecta de todos los tipos de .NET, tanto de los existentes como de los que se crean nuevos (todo hereda de System.Object!). La herencia de *Object* es implícita y, por tanto, su declaración es inútil. Todos los tipos heredan de sus métodos e incluso pueden sustituir algunos.

Esto es lo que hace *System.ValueType* que, en la herencia de tipos de .NET, se convierte en la base de la familia "Valores", adaptándose a los métodos de System.Object.

**Métodos de object**

public bool Equals(object)

protected void Finalize()

public int GetHashCode()

public System.Type GetType()

protected object MemberwiseClone()

public void Object()

public string ToString()

# **CLASES Y OBJETOS EN POO**

## **Clase ó Plantilla**

Una clase es una construcción que permite crear tipos personalizados propios mediante la agrupación de variables de otros tipos, métodos y eventos. Una clase es como un plano, un molde. Define los datos y el comportamiento del tipo personalizado. La variable permanece en memoria hasta que todas las referencias a ella estén fuera de alcance. En ese momento, CLR la marca como apta para la recolección de elementos no utilizados (tarea que realiza el recolector de basura “Garbage Collector”). Si la clase no se declara como estática, el código de cliente puede utilizarla mediante la creación de objetos o instancias que se asignan a una variable. Si la clase se declara como estática (static), solo existe una copia en memoria y el código de cliente solo puede tener acceso a ella a través de la propia clase y no de una variable de instancia.

### **Declarar clases**

Las clases se declaran mediante la palabra clave **class**, tal como se muestra en el ejemplo siguiente:

public class Cliente  
 {  
 //Los campos, propiedades, métodos y eventos van aquí  
 }

El modificador de acceso precede a la palabra clave **class**. Como, en este caso, se utiliza **public**, cualquiera puede crear/instanciar objetos de esta clase. Los modificadores de acceso indican la visibilidad de la clase respecto de quién quiera usarla, es decir de la **variables de instancia** que existan de esta clase. El nombre de la clase sigue a la palabra clave **class**. El resto de la definición es el cuerpo de clase, donde se definen el comportamiento y los datos.

### **Miembros**

Los campos, propiedades, métodos y eventos de una clase se conocen colectivamente como ***miembros de clase***.

### **Crear objetos**

Aunque se las nombra a veces como si fueran lo mismo, **una clase y un objeto son cosas diferentes**. Una clase define un tipo de objeto, pero no es propiamente un objeto. Un objeto es una entidad concreta basada en una clase, **un objeto es la instancia de una clase**.

Los objetos se pueden crear con la palabra clave **new** seguida del nombre de la clase en la que se basará el objeto, de la siguiente manera:

Cliente nuevoCliente = new Cliente();

En el ejemplo anterior, **nuevoCliente** es una referencia a un objeto basado en la plantilla Cliente. Es una variable de instancia. Esta referencia apunta al nuevo objeto, pero no contiene los datos del objeto, es decir, la referencia contiene la dirección de memoria donde estarán los datos del objeto.

De hecho, se puede crear una referencia a un objeto sin crear el objeto:

Cliente otroCliente;

No se recomienda crear este tipo de referencias que realmente no apuntan a un objeto existente, ya que al intentar el acceso a un objeto a través de esa referencia se producirá un error en tiempo de ejecución. No obstante, este tipo de referencia se puede crear para hacer referencia a un objeto, ya sea creando un nuevo objeto o asignándola a un objeto existente, de la siguiente forma:

Cliente objeto3 = new Cliente();  
 Cliente objeto4 = objeto3;

Este código crea dos variables con referencias a objeto que se refieren al mismo objeto. Por consiguiente, los cambios realizados en el objeto a través de **objeto3** se reflejarán en los usos posteriores de **objeto4**. Puesto que el acceso a los objetos basados en clases se realiza por referencia, las clases se denominan tipos por referencia.

# **PROPIEDADES DE LA POO**

## 

## **Herencia de clase**

Haremos una breve introducción sobre Herencia en POO. Luego profundizaremos este tema en la siguiente clase del curso.

La herencia se realiza a través de una *derivación*, lo que significa que una clase se declara utilizando una *clase base* de la cual hereda los datos y el comportamiento. Una **clase base** se especifica anexando dos puntos y el nombre de la clase base a continuación del nombre de la **clase derivada**; en el ejemplo, la clase Jefe deriva de la clase base Empleado del siguiente modo:

public class Jefe : Empleado  
 {  
 // los campos, propiedades, métodos y eventos de la clase base Empleado los heredará la clase Jefe  
 // Jefe aparte podrá tener sus propios campos, propiedades, métodos y eventos

}

Cuando una clase deriva de una clase base, hereda todos los miembros de la **clase base** excepto los constructores.

A diferencia de C++ que es un lenguaje de programación orientado a objetos puro, una clase en C# solo puede heredar directamente de una sola clase base. Sin embargo, dado que una clase base puede heredar de otra clase, una clase puede heredar indirectamente de varias clases base. Además, una clase puede implementar directamente más de una interfaz. Heredando de una clase base y de las interfaces necesarias, se logra la herencia múltiple.

Una clase se la puede declarar como abstracta. Una clase abstracta (abstract) contiene métodos abstractos que incluyen una definición de firma pero ninguna implementación. No se pueden crear instancias de las clases abstractas. Solo se pueden utilizar a través de clases derivadas que implementan los métodos abstractos. Por el contrario, una clase sellada (sealed) no permite que otras clases deriven de ella.

## **Propiedades**

Una propiedad es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Se pueden usar las propiedades como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados *descriptores de acceso*. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

En este ejemplo, la clase **PeriodoDeTiempo** almacena un período de tiempo. Internamente, la clase almacena el tiempo en segundos, pero una propiedad denominada **Horas** permite especificar el tiempo en horas. Los descriptores de acceso para la propiedad **Horas** realizan la conversión entre horas y segundos.

class PeriodoDeTiempo  
 {  
 private double segundos;  
  
 public double Horas  
 {  
 get { return segundos / 3600; }  
 set { segundos = value \* 3600; }  
 }  
 }  
  
   
 class Program  
 {  
 static void Main()  
 {  
 PeriodoDeTiempo t = new PeriodoDeTiempo();  
  
 // La asignación de la propiedad Horas provoca la llamada del descriptor de acceso 'set'.  
 t.Horas = 24;  
  
 // La evaluación de la propiedad Horas provoca la llamada del descriptor de acceso 'get'.  
 System.Console.WriteLine("Tiempo en horas: " + t.Horas);  
 }  
 }



## **Definiciones de cuerpos de expresión**

## 

Es habitual tener propiedades que simplemente devuelvan de inmediato el resultado de una expresión. Hay un acceso directo de sintaxis para definir estas propiedades mediante =>, se lo llama operador lambda:

public string NombreCompleto => Nombre + " " + Apellido;

La propiedad debe ser de solo lectura, y no se utiliza la palabra clave de descriptor de acceso get.

## 

## **Propiedades autoimplementadas**

En C# 3.0 y versiones posteriores, las propiedades implementadas automáticamente hacen que la declaración de propiedades sea más concisa cuando no es necesaria ninguna lógica adicional en los descriptores de acceso de la propiedad. También permite que el código cree objetos. Cuando se declara una propiedad tal como se muestra en el ejemplo siguiente, el compilador crea un campo de respaldo privado y anónimo al que solo se puede acceder con los descriptores de acceso de propiedad get y set.

En el ejemplo siguiente se muestra una clase simple que tiene algunas propiedades implementadas automáticamente:

## 

class Cliente

{

// Propiedades autoimplementadas

public double TotalCompras { get; set; }

public string Nombre { get; set; }

public int ClienteID { get; set; }

// Constructor

public Cliente(double compras, string nombre, int ID)

{

TotalCompras = compras;

Nombre = nombre;

ClienteID = ID;

}

// Métodos

public string RecuperarInfoDeContacto() { return "Información de contacto del cliente"; }

public string RecueprarHistorialDeTransacciones() { return "Historial de compras del cliente"; }

// .. Más métodos, eventos, etc., de la clase Cliente.

}

class Program

{

static void Main()

{

// Creación de un nuevo objeto cliente, llamando al constructor sobrecargado.

Cliente clie = new Cliente(4987.63, "El Trevol", 90108);

// Modificación de la propiedad autoimplementada TotalCompras

clie.TotalCompras += 499.99;

}

}

En C# 6 y versiones posteriores, puede inicializar las propiedades implementadas automáticamente de forma similar a los campos:

## public string Nombre { get; set; } = "María";

## 

## **Información general sobre propiedades**

* Las propiedades permiten que una clase exponga una manera pública de obtener y establecer valores, a la vez que se oculta el código de implementación o comprobación.
* Se utiliza un descriptor de acceso de propiedad **get** para retornar el valor de propiedad y un descriptor **set** para asignarle un nuevo valor a la propiedad. Estos descriptores de acceso pueden tener diferentes niveles de acceso, es decir distinta visibilidad.
* La palabra clave **value** se utiliza para definir el valor asignado por el descriptor de acceso **set**. En definitiva el valor que el usuario ingrese en dicho campo llega como parámetro a la propiedad con la palabra clave **value**.
* Las propiedades que no implementan un descriptor de acceso **set** son de solo lectura. También pueden tener un descriptor **set** privado y de esta manera ser de solo lectura. Otra manera de emular este comportamiento es declarar el miembro con la visibilidad **readonly**, de esta manera los valores de la variable no podrán cambiar en ningún hilo de ejecución (thread) de la aplicación.
* Cuando se trate de propiedades sencillas que no requieran ningún código de descriptor de acceso personalizado, considere la posibilidad de utilizar propiedades implementadas automáticamente.

## **Métodos, firmas de método y sobrecargas**

Los métodos definen la funcionalidad de la clase, y se definen como una función de C#.

Se declaran en una clase o struct especificando el nivel de acceso, como public o private, y modificadores opcionales como abstract o sealed, el valor de retorno, el nombre del método y cualquier parámetro de método. Todas estas partes de la declaración del método forman la **firma del método**.

En el siguiente ejemplo el método se llama MostrarNombreCompleto, y su firma está formada el nivel de acceso **public**, el valor de retorno **string**, y los dos parámetros **nombre** y **apellido**, ambos tipo **string**:

public string MostrarNombreCompleto(string nombre, string apellido)

{

return nombre + ' ' + apellido;

}

Esta firma puede ser diferente en sus tipos, dado que no se reconoce el nombre de cada variable como parte de la firma sino más bien los tipos que recibe. Pueden existir entonces infinitos Métodos o funciones con el mismo nombre siempre y cuando la cantidad y orden de los tipos de dato que reciba sean diferentes.

Sobrecargando el método anterior para que reciba la inicial del segundo nombre, cambia la firma a el nivel de acceso **public**, el valor de retorno **string**, y los tres parámetros **nombre, otroNombre** y **apellido**, tipos **string, char, string**:

public string MostrarNombreCompleto(string nombre, char otroNombre, string apellido)

{

return nombre + ' ' + otroNombre + ' ' + apellido;

}

No obstante para la sobrecarga se necesita una combinación de nombre de clase con parámetros diferentes. Mismo nombre de clase con mismos parámetros no compila aunque el valor de retorno sea diferente. Aunque sea parte de la firma.

Los parámetros de un método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros. Esta clase contiene tres métodos con código propio definido y uno abstracto a ser implementado por la clase que herede de la clase Moto:

abstract class Moto  
 {  
 // Este método se puede llamar de cualquier lugar porque es public.  
 public void IniciarMotor() {/\* Aquí iría el código \*/ }  
  
 // Solo clases derivadas pueden ver este método porque es protected

protected void CargarNafta(int litros) {/\* Aquí iría el código \*/ }  
  
 // Clases derivadas pueden sobrescribir el código de la clase base porque es virtual.  
 public virtual int Conducir(int kilómetros, int velocidad)

{/\* Aquí iría el código \*/return 1; }  
  
 // Las clases derivadas pueden implementarlo.  
 public abstract double ObtenerVelMáxima();   
 }

# **Constructores**

Cada vez que se crea una clase o struct, se llama a su constructor. El constructor es un método que se llama cuando se crea la clase o struct. Una clase o struct puede tener varios constructores que toman argumentos diferentes, esto se denomina sobrecarga de constructores. Los constructores permiten al programador establecer valores predeterminados, limitar la creación de instancias y escribir código flexible y fácil de leer.

Si no proporciona un constructor para el objeto, C# creará uno de forma predeterminada que cree instancias del objeto y establezca las variables miembro en los valores predeterminados.

### **Constructores de Instancia**

Los constructores de instancias se usan para crear e inicializar cualquier variable de miembro de instancia cuando se usa la expresión **new** para crear un objeto de una clase.

En el ejemplo vemos la clase **Coordenadas** con el constructor por omisión, en el cual se le agregó la inicialización de las variables x e y:

class Coordenadas

{

public int x, y;

// constructor

public Coordenadas ()

{

x = 0;

y = 0;

}

}

### **Constructor con dos argumentos**

Aquí sobrecargamos el constructor del ejemplo anterior, teniendo otro que recibe dos argumentos.

public Coordenadas (int x, int y)

{

this.x = x;

this.y = y;

}

Esto permite crear objetos Coordenadas con valores iniciales concretos o predeterminados, del siguiente modo:

Coordenadas p1 = new Coordenadas ();  
 Coordenadas p2 = new Coordenadas (5, 3);

Si una clase no tiene un constructor, se genera automáticamente un constructor predeterminado y se usan valores predeterminados para inicializar los campos de objeto. Por ejemplo, un int se inicializa a 0. Por consiguiente, como el constructor predeterminado de la clase Coordenadas inicializa todos los miembros de datos en cero, se puede quitar del todo sin cambiar el funcionamiento de la clase. Los constructores de instancia también se pueden utilizar para llamar a los constructores de instancia de clases base. El constructor de clase puede invocar el constructor de la clase base a través del inicializador, en el siguiente ejemplo, el constructor de la clase Circulo que hereda de la clase Figura llama al constructor de su clase base que es Figura:

class Circulo : Figura  
 {  
 public Circulo (double radio)  
 : base(radio, 0)  
 {  
 }  
 }

# **TOSTRING()**

**Object.ToString** es el principal formato de método en .NET Framework. Convierte un objeto en su representación de cadena de caracteres así será apropiado para su presentación. En las implementaciones donde no se tenga un método **ToString** de forma predeterminada el método **Object.ToString** devuelve el nombre completo incluyendo el namespace del tipo de objeto.