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 las **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.