**POO en C#**

***CLASES***

Una clase se declara utilizando la palabra clave **class** seguida de un nombre y de un bloque de código delimitado por los caracteres **{}**.

**Sintaxis:**

<modificador de acceso> [partial] class <nombre de la clase>:

[<Clase base>, <Interfaz1>, <Interfaz2>, ...]

{

}

**Modificadores de acceso:**

private: este modificador es válido únicamente en el caso de clases anidadas, y en ese caso la clase es visible únicamente por el tipo que la contiene (consulte el siguiente ejemplo).

protected: este modificador es válido únicamente en el caso de clases anidadas, y en ese caso la clase es visible únicamente por el tipo que la contiene y sus tipos derivados (consulte el siguiente ejemplo).

internal: la clase es visible por todas las clases definidas en el ensamblado.

protected internal: la clase es visible por todas las clases definidas en el ensamblado y por una clase exterior al ensamblado únicamente si hereda de ella.

public: la clase es visible por cualquier otro tipo, sin importar a qué ensamblado pertenezca.

**Constructor:**

Un constructor es un método especial que permite instanciar un objeto. Su nombre es idéntico al de la clase en la que se ha definido, y no tiene ningún tipo de retorno.

**Sintaxis del constructor:**

<modificador de acceso> <Nombre de la clase>([<parámetro1>,

<parámetro2>...])

{

}

**Sintaxis de clase con sobrecarga de constructores:**

public class Empleado

{

   private int edad;

   private int salario

   public Empleado()

   {

   }

   public Empleado(int edadEmpleado)

   {

         edad = edadEmpleado;

   }

   public Empleado(int edadEmpleado, int salarioEmpleado)

   {

         edad = edadEmpleado;

         salario = salarioEmpleado;

   }

}

C# Permite el uso de **=>** para declaración de métodos o de sentencias en una sola línea. Ejemplo:

public Empleado(int edadEmpleado)

        => edad = edadEmpleado;

En vez de.

public Empleado(int edadEmpleado)

{

edad = edadEmpleado;

}

**Destructor:**

La destrucción de un objeto es una operación automática en una aplicación .NET. El Garbage Collector (GC) supervisa la totalidad de instancias creadas a lo largo de la ejecución de la aplicación y marca como huérfano cada objeto que no se utiliza. Cuando el sistema necesita más memoria, el GC destruye aquellos objetos que considera que es posible destruir invocando a sus destructores, si los tienen.

**Sintaxis de destructor:**

~Coche()

{

   //Incluir aquí el código que permite limpiar

   //los recursos utilizados por el objeto

}

O también:

~Coche() => <instrucción única>;

**Clases parciales:**

Con las clases parciales es posible dividir una clase en varias porciones que pueden estar repartidas en varios archivos.

Es posible declarar una clase parcial de la misma manera que una clase clásica, pero la palabra clase class debe estar prefijada por la palabra clave partial. Todas las partes de una clase parcial deben declararse con el mismo nombre en el mismo ensamblado y en el mismo espacio de nombres. Si no se respetan estas condiciones, el compilador considerará las distintas partes como clases diferentes, lo cual, evidentemente, no coincide con el resultado esperado.

**Ejemplos de clases parciales:**

public partial class Empleado

{

   public int salario;

}

public partial class Empleado

{

   public int edad;

}

A la final se compilará una clase Empleado con los atributos de salario y edad. Podrá ser utilizada así:

//La variable empleado es de tipo Empleado

empleado.edad = 19;

empleado.salario = empleado.salario \* 1.1;

***ESTRUCTURAS***

Las estructuras son, como las clases, conjuntos coherentes de datos y funcionalidades. Son similares a las clases, pero tienen como principal diferencia que son tipos valor.

Esta diferencia implica comportamientos diferentes, en particular el hecho de que un objeto de tipo estructura no puede ser null y tiene, obligatoriamente, un valor. Tras la declaración de una variable de tipo estructura, cada uno de los miembros del objeto está inicializado a su valor por defecto.

**Sintaxis:**

<modificador de acceso> [partial] struct <nombre de la estructura>

{

}

**Ejemplo:**

public struct Coche

{

   public int kilometraje;

}

Coche miFerrari;

miFerrari.kilometraje = 42;

***METODOS***

Un método es una función o un procedimiento definido en una clase o una estructura. Define una funcionalidad asociada al objeto y permite manipular variables del objeto para modificar su estado.

**Creación:**

Para definir un método, basta con escribir un procedimiento o una función en el interior de una definición de tipo. Ejemplo:

public class Coche

{

   private bool estáArrancado;

   public void Arrancar()

   {

          estáArrancado = true;

          Console.WriteLine("RUN RUN");

   }

}

También se le puede aplicar la sintaxis simplificada:

public void Arrancar() => estáArrancado = true;

En esta sintaxis simplificada hay que tener en cuenta una sutileza. Cuando una función (que devuelve un valor) se reescribe de esta manera, la palabra clave return no debe aparecer.

Una vez definido, un método puede, como una variable, invocarse sobre un objeto gracias al operador punto (.), siempre que el modificador de acceso del método lo autorice.

miFerrari.Arrancar();

Varios métodos pueden tener el mismo nombre, pero deben diferenciarse por sus argumentos. Se dice, en tal caso, que los métodos están sobrecargados.

public void EnviarCorreo(string nombreDestinatario, Dirección

direcciónDestinatario)

{

}

public void EnviarCorreo(Dirección direcciónDestinatario, string

nombreDestinatario)

{

}

public void EnviarCorreo(string nombreDestinatario, Dirección

direcciónDestinatario, decimal precioSello)

{

}

public void EnviarCorreo(string nombreDestinatario, Dirección

direcciónDestinatario, string direcciónRemitente)

{

}

**Métodos parciales:**

El principio de los métodos parciales es relativamente simple: una parte del método se define con la palabra clave partial pero no posee bloque de código asociado. Esto permite al compilador saber que el método es susceptible de tener una implementación en otra parte de la clase.

Los métodos parciales deben estar definidos en clases parciales. En caso contrario, el compilador genera un error.

Los métodos parciales no pueden ser funciones. En efecto, el código que invoca a esta función dependería de ellas para poder funcionar correctamente, lo que no se corresponde con la lógica de extensión de funcionalidad descrita antes.

Por el mismo motivo no pueden tener modificadores de acceso, y su visibilidad se define directamente como private por parte del compilador.

partial class Coche

{

   partial void EnArranque();

   void Arrancar()

   {

          EnArranque();

   }

}

La segunda parte del método puede definirse en el mismo archivo o en otro archivo diferente, pero siempre dentro de la misma clase parcial. Debe tener la misma firma que la primera parte.

partial class Coche

{

   partial void EnArranque()

   {

          Console.WriteLine("RUN RUN");

   }

}

Si la segunda parte del método no está presente, entonces el método parcial no tiene implementación. En este caso, el compilador suprime la definición del método parcial así como las llamadas a dicho método.

**Métodos de extensión:**

Los métodos de extensión permiten agregar funcionalidades a las clases o a las estructuras existentes sin modificar su código. Este tipo de método está adaptado a la extensión de clases del framework .NET tales como la clase String. En efecto, estas clases no pueden modificarse y ocurre con frecuencia que se desea modificar su comportamiento agregando alguna funcionalidad. Como los métodos utilidades descritos anteriormente, los métodos de extensión se escriben en el exterior de la clase extendida pero pueden utilizarse de la misma manera que los métodos de la clase extendida.

Ejemplo:

**static** class Extensions

{

   public **static** bool EsPalíndromo(**this string** cadenaAComprobar)

   {

          if (cadenaAComprobar == null)

                 return false;

          string cadenaInversa = "";

          for (int i = cadenaAComprobar.Length - 1; i >= 0; i--)

          {

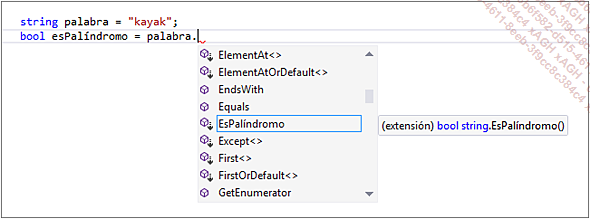
               cadenaInversa = cadenaInversa + cadenaAComprobar [i];

          }

          return cadenaInversa.Equals(cadenaAComprobar);

   }

}

Si utilizamos, a continuación, una variable de tipo string, el método EsPalíndromo está disponible y lo sugiere IntelliSense.

Los métodos de extensión se recuperan en la lista con un icono diferente a los métodos clásicos así como con la precisión "(extension)" en el texto informativo asociado.

**Métodos operadores:**

Los métodos operadores permiten sobrecargar un operador estándar del lenguaje para utilizarlo sobre tipos definidos por el desarrollador.

En el caso de dos colecciones de sellos, sobrecargar el operador + permitiría fusionar las colecciones.

Tomemos dos objetos de tipo ColecciónDeSellos definidos como se muestra a continuación y probemos a sumarlos:

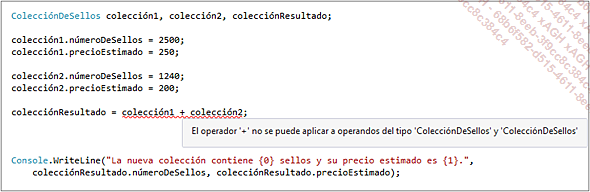
struct ColecciónDeSellos

{

   public int númeroDeSellos;

   public int precioEstimado;

}



El compilador no sabe cómo sumar ambos elementos y nos lo indica mostrando un mensaje de error bloqueante para la compilación. Si sobrecargamos el operador + para nuestra estructura:

public **static** ColecciónDeSellos **operator +**(ColecciónDeSellos

colección1, ColecciónDeSellos colección2)

{

   ColecciónDeSellos resultado;

   resultado.numeroDeSellos = colección1.numeroDeSellos +

colección2.numeroDeSellos;

   resultado.precioEstimado = colección1.precioEstimado +

colección2.precioEstimado;

   return resultado;

}

Una vez agregado este código, el error de compilación desaparece y la ejecución del código muestra el siguiente resultado:

La nueva colección contiene 3740 sellos y su precio

estimado es de 450€

***PROPIEDADES***

Las clases pueden contener datos públicos con la forma de variables miembro, pero para respetar el concepto de encapsulación, es preferible exponer propiedades en lugar de variables. En efecto, las propiedades permiten ejecutar código con cada acceso de lectura o de escritura. Esto permite validar los datos y, de este modo, mantener el objeto en un estado coherente.

**Lectura y escritura:**

Una propiedad se asemeja a una declaración de función pero contiene dos bloques de código identificados por las palabras clave get y set. El bloque get se ejecuta con la lectura de la propiedad mientras que el bloque set se ejecuta cuando se accede a la propiedad en escritura.

Una propiedad tiene el siguiente aspecto:

private int kilometraje;

public int Kilometraje

{ 

**get**

   {

       return kilometraje;

   } 

**set**

   {

       if (value > kilometraje)

       {

           kilometraje = value;

       }

       else

       {

           Console.WriteLine("El kilometraje de un coche no

puede disminuir");

       }

   }

}

Para respetar el principio de encapsulación la variable kilometraje se marca como privada con private.

Podríamos destacar la palabra clave value que encontramos en el bloque set. Representa al valor asignado a la propiedad. Se realiza una comprobación en el bloque set para validar el valor que se pasa como argumento a la propiedad.

Los bloques get y set de las propiedades se pueden utilizar con una sintaxis simplificada cuando tienen una sola instrucción. Se puede ver que en el bloque get se omite la palabra clave return: está presente de modo implícito ya que el bloque debe devolver un valor.

private int kilometraje;

public int Kilometraje

{

**get** => kilometraje;

**set** => kilometraje = value;

}

**Solo lectura:**

Puede resultar interesante, para la coherencia de un objeto, exponer una propiedad únicamente en solo lectura. En nuestro caso, podríamos decir que el kilometraje de un vehículo no debería modificarlo más que el funcionamiento interno del mismo. Proteger la variable de instancia kilometraje exponiéndola únicamente en solo lectura parece, en este caso, una excelente solución.

Para ello, basta con declarar la propiedad sin bloque set. Sin este bloque, cualquier intento de modificar el valor de la propiedad produce un error de compilación.

private int kilometraje;

public int Kilometraje

{

   get => kilometraje;

}

}

En casos muy asilados, podría resultar útil definir una propiedad de solo escritura. Por ejemplo, podríamos querer definir una propiedad Contraseña en una clase Usuario. Es evidente que no queremos que este campo pueda leerse. Una propiedad de solo escritura parece, por tanto, una solución adecuada para implementar este comportamiento.

Una propiedad de solo escritura se implementa omitiendo el bloque get.

private string contraseña;

public string Contraseña

{

   set => contraseña = value;

   }

}

**Propiedades automáticas:**

Cuando no se asocia ninguna lógica particular a los métodos de acceso get y set de una propiedad, puede resultar interesante utilizar las propiedades automáticas. Estos elementos se declaran mediante un atajo sintáctico para la definición de las propiedades.

Una propiedad automática se declara como una propiedad clásica, pero no se facilita el cuerpo de los bloques get y set. Agreguemos una propiedad Color a nuestro Coche:

public string Color { get; set; }

Como esta propiedad no accede explícitamente a una variable miembro, no es necesario declarar una para almacenar el valor asignado. Entre bastidores, el compilador crea una y genera el cuerpo de los bloques get y set. A continuación se muestra una representación en C# del código IL generado por el compilador.

private string color ;

public string Color

{

get => return color;

set => color = value;

}

Uno u otro de los bloques get y set puede estar marcado por un modificador para proteger el acceso a la propiedad (y solo exponer públicamente la propiedad en solo lectura, por ejemplo).

public string Color { get; private set; }

**Inicialización de propiedades automáticas:**

public string Color { get; set; } = "Verde";

public string Color { get; set; } = EncontrarNombreColorRGB(0, 255, 0);

**Propiedades automáticas de solo lectura:**

La posibilidad de inicializar una propiedad en el momento de su declaración aporta a C# la capacidad de crear propiedades automáticas de solo lectura. Sin inicialización, una propiedad automática de solo lectura no tendría, en efecto, ninguna utilidad.

Este tipo de propiedad se declara de la misma forma que una propiedad automática con inicialización. La única diferencia es la eliminación de la palabra clave set para obtener una propiedad de solo lectura. La propiedad Color definida más arriba puede declararse de solo lectura de la siguiente manera:

public string Color { get; } = "Verde";

**Propiedades indexadas:**

Las propiedades indexadas, también llamadas indexadores o propiedades por defecto, permiten realizar un acceso de tipo array a un conjunto de elementos.

Estas propiedades reciben uno o varios parámetros que se pasan entre corchetes. No tienen nombre y una clase puede poseer varias propiedades indexadas que deben diferenciarse por sus parámetros.

Una propiedad indexada se declara utilizando la siguiente sintaxis:

<modificador de acceso> <tipo> this[parámetro1, [parámetro2,

parámetro3...]]

{

   //Accesores get y/o set

}

Una implementación (totalmente inútil) de una propiedad indexada podría ser la siguiente:

public class Coche

{

   public string **this[int numeroRueda]**

   {

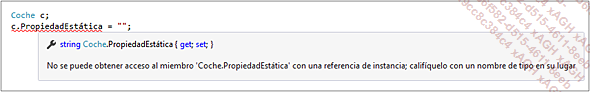
         get => return "Rueda n°" + numeroRueda;

   }

}

**Miembros estáticos:**

Los miembros estáticos (o miembros compartidos) de una clase son variables, propiedades o métodos que no son específicos de una instancia de la clase, sino de la propia clase. Un miembro estático existe en un único ejemplar. Acceder a este miembro a través de un objeto no tendría sentido, por lo que se accede mediante el nombre del tipo asociado. Intentar utilizarlo desde una instancia produce un error de compilación.



Un miembro estático no puede tampoco acceder a un miembro de instancia de su clase, salvo si se le provee el objeto del que se quiere utilizar el miembro estático.

Es posible definir un miembro compartido utilizando la palabra clave static tras el modificador de acceso.

Aquí, consideramos que el número de ruedas es una característica idéntica para todos los coches. Es, por tanto, posible agregar una propiedad estática que defina este número en la clase Coche:

public static int NumeroDeRuedas get => 4;

Para utilizar esta propiedad, escribiremos:

int numero = Coche.NumeroDeRuedas;

**Uso de clases y estructuras:**

**Instancias anónimas:**

En ciertos casos cuando se necesita trabajar con atributos de una clase que por mucho se utilizará dos veces, se crean clases anónimas, las cuales se crean en el momento de su inicialización y son poco duraderas, ésas no tienen, o deberían tener, ninguna influencia importante en el programa.

Para declarar una clase anónima basta con declarar atributos dentro de {} y antepuestos de la palabra new:

**var** coche =  **new {** Color = "Rojo", Kilometraje = 12500 **}**