UNIDAD 9. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

1. Modularizando y organizando nuestras clases

1.1. Espacio de nombres

Del mismo modo que los ficheros se organizan en directorios, los tipos de datos se organizan en espacio de nombres. Por un lado estos espacios permiten tener más organizados los tipos de datos, lo que facilita su localización. De hecho, esta es la forma en que se encuentra organizada la BCL, de modo que todas las clases más comúnmente usadas en cualquier aplicación pertenecen al espacio de nombres llamado System, las de acceso a bases de datos en System.Data, las de realización de operaciones de entrada/salida en System.IO, etc

Sentencia using -> En principio, si desde código perteneciente a una clase definida en un cierto espacio de nombres se desea hacer referencia a tipos definidos en otros espacios de nombres, se ha de referir a los mismos usando su nombre completamente calificado. Como puede resultar muy pesado tener que escribir nombres tan largos en cada referencia a tipos así definidos, en C# se ha incluido un mecanismo de importación de espacios de nombres que usa la siguiente sintaxis:

```
using <espacioNombres>;
```

Este tipo de sentencias siempre ha de aparecer dentro de una definición de espacio de nombres antes que cualquier definición de miembros de la misma y permite indicar cuáles serán los espacios de nombres que se usarán implícitamente dentro de ese espacio de nombres. A los miembros de los espacios de nombres así importados se les podrá hacer referencia sin tener que usar calificación completa.

Definición del espacio de nombres –> Para definir un espacio de nombres se utiliza la siguiente sintaxis:

```
namespace <nombreEspacio>
{
        <tipos>
}
```

Los tipos que se definan en <tipos> pasarán a considerase pertenecientes al espacio de nombres llamado <nombreEspacio>. Como veremos más adelante, aparte de clases esto tipos pueden ser también interfaces, estructuras, tipos enumerados y delegados. A continuación se muestra un ejemplo en el que definimos una clase de nombre **ClaseEjemplo** perteneciente a un espacio de nombres llamado **EspacioEjemplo**:

```
namespace EspacioEjemplo
{
   class ClaseEjemplo { }
}
```

El verdadero nombre de una clase, al que se denomina nombre completamente cualificado, es el nombre que le demos al declararla prefijado por la concatenación de todos los espacios de nombres a los que pertenece ordenados del más externo al más interno y seguido cada uno de ellos por un punto (carácter .) Por ejemplo, el verdadero nombre de la clase **ClaseEjemplo** antes definida es **EspacioEjemplo.ClaseEjemplo**.

Aparte de definiciones de tipo, también es posible incluir como miembros de un espacio de nombres a otros espacios de nombres. Es decir, como se muestra el siguiente ejemplo es posible anidar espacios de nombres:

```
namespace EspacioEjemplo
{
  namespace EspacioEjemplo2
  {
    class ClaseEjemplo  {}
  }
}
```

Ahora **ClaseEjemplo** tendrá **EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo** como nombre completamente calificado. En realidad es posible compactar las definiciones de espacios de nombres anidados usando esta sintaxis de calificación completa para dar el nombre del espacio de nombres a definir. Es decir, el último ejemplo es equivalente a:

```
namespace EspacioEjemplo.EspacioEjemplo2
{
     class ClaseEjemplo   {}
}
```

Alias de espacio de nombres y de tipos -> Con la sentencia using también podemos crear alias que nos permiten facilitar el acceso a espacios de nombres o a Tipos. Para ello debemos asignarle a un identificador el espacio de nombres o tipo al que nos queramos referir, esto lo haremos de la siguiente manera:

```
using identificador=EspaciodeNombres1.EspaciodeNombres2;
using identificador=EspaciodeNombres.Class;
```

```
using System;
using System.Text;
using EEC = EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo;
using EE = EspacioEjemplo.EspacioEjemplo2;
```

```
namespace EspacioEjemplo
{
    namespace EspacioEjemplo2
    {
        class ClaseEjemplo
        {
            public static void Saludo()
            {
                Console.WriteLine("Hola Mundo");
            }
}}}

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo.Saludo(); //Sin
                                                Alias
            EE.ClaseEjemplo.Saludo(); //Utilizando el alias para los
                                            dos espacios
            EEC.Saludo(); //Utilizando el alias completo
        }
}}
```

1.2. Librerías

Además de por espacios de nombres, también podremos separar nuestras Clases, en unidades mayores denominadas librerías. En el caso de C# son ficheros que contendrán MSIL de clases organizadas en uno o más espacios de nombres o paquetes. Dependiendo el lenguaje de programación, se generaran distintos tipos de archivos.En lenguajes compilados a bytecode. Estás librerías podrán ser por ejemplo:

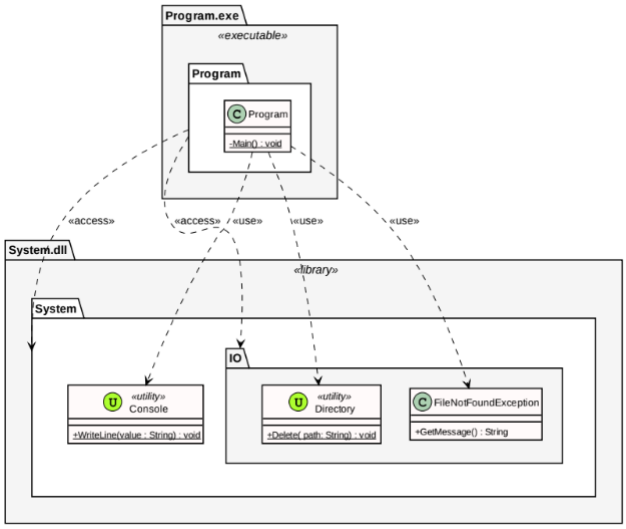
- Ensamblados con extensión .DLL en para C#, VB.net, F# en Windows.
- Artefactos con extensión .JAR en el caso de Java, Kotlin, Sacala y Groovy.

Se pueden publicar las librerías opensource y usar librerías de terceros a través de gestores de paquetes como:

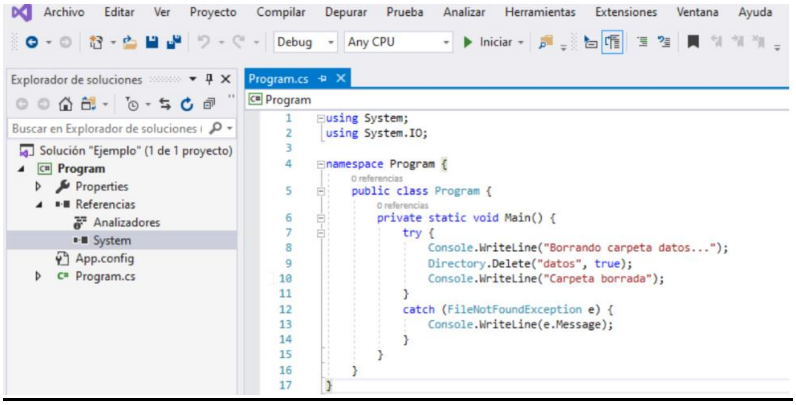
- Nuget para ensamblados de C#, F# y VB.net.
- Maven para artefactos de Java, Kotlin, Scala, Groovy.

Concepto de dependencia → Hasta ahora en todas las aplicaciones que hemos realizado siempre hemos usado librerías, pero solamente han sido las librerías de clases incluidas en la BCL. Si suponemos un programa sencillo en el que hayamos utilizado, por ejemplo, la clase Console incluida en el espacio de nombres System o la clase Directory incluida en System.IO. Realmente lo que está ocurriendo es que **se ha generado una dependencia entre clases**, ya que nuestra clase Program usa otras clases como Console y Directory. Además tendremos una **dependencia entre librerías o ensamblados**. Ya que las clases que usamos están definidas en los paquetes System y System.IO dentro de System.dll.

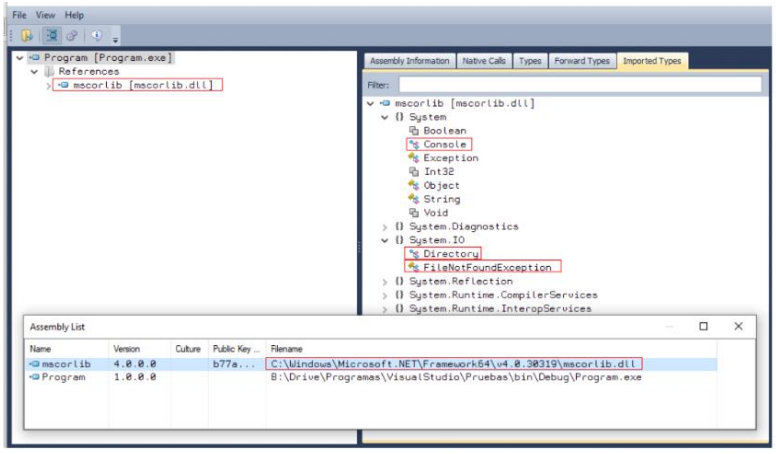
```
using System;
using System.IO;
namespace Program
{
  public class Program
  {
    private static void Main()
    {
      try
      {
        Console.WriteLine("Borrando carpeta
                           datos...");
        Directory.Delete("datos", true);
        Console.WriteLine("Carpeta borrada");
      }
      catch (FileNotFoundException e)
      {
         Console.WriteLine(e.Message);
      }
    }
 }
}
```



Podemos ver y establecer las dependencias de nuestro programa a través del apartado referencias en nuestro proyecto.

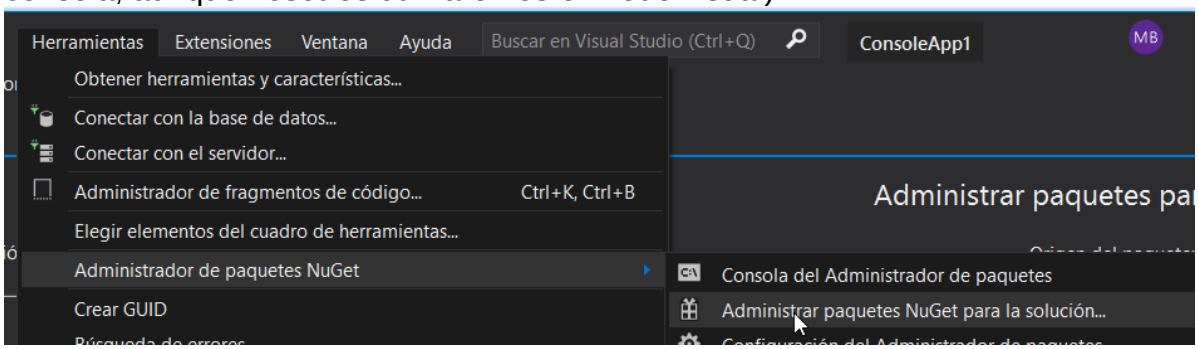


Además, podremos consultar las dependencias de nuestros programas con herramientas externas como [.NET Dependency Walker](). Si examinamos Program.exe:

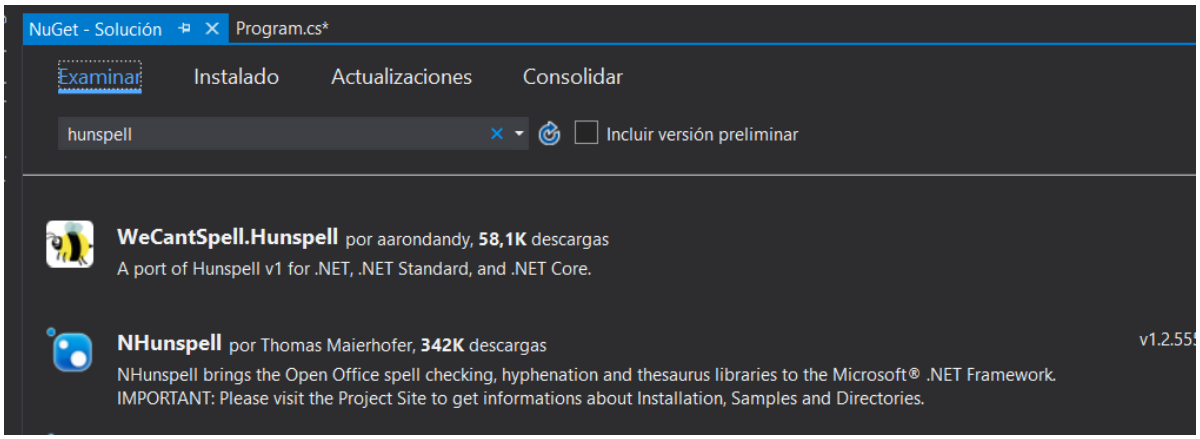


Ejercicio ejemplo para usar librerías de terceros:

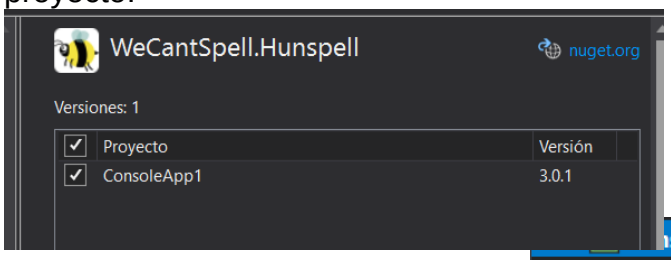
En nuestro caso queremos comprobar ortográficamente una palabra introducida por teclado. Crearemos un proyecto e iremos al administrador de paquetes NuGet, lo podemos encontrar dentro de la pestaña Herramientas (se puede abrir en modo consola, aunque nosotros utilizaremos el modo visual).



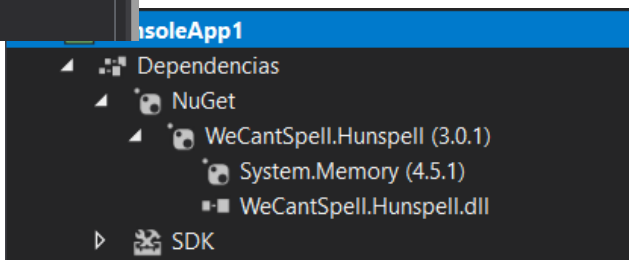
Deberemos indicar el nombre o parte de este, para que NuGet pase a realizar la búsqueda de los paquetes que correspondan con la descripción introducida. En nuestro caso Hunspell.



Seleccionaremos WeCantSell.Hunspell y le indicaremos que se instale en el proyecto.



Tras hacerlo comprobaremos que un montón de referencias nuevas se han añadido a nuestro proyecto.



Nos bajaremos del FTP el diccionario Spanish.dic y Spanish.aff, que previamente hemos obtenido del repositorio de diccionarios para Hunspell en

https://github.com/titoBouzout/Dictionaries y lo copiaremos en la carpeta donde se genera nuestro ensamblado.
Añadiremos el código que se muestra más abajo en nuestro programa principal.

```
using System;
using System.Linq;
using WeCantSpell.Hunspell;
public class Program
{
    static void Main()
    {
        Console.Write("Cargando diccionario...\n");
        WordList diccionario = WordList.CreateFromFiles("Spanish.dic");
        string palabra;
        do
        {
            Console.Write("Introduce una palabra: ");
            palabra = Console.ReadLine();
            string mensaje;
            if (!diccionario.Check(palabra))
            {
                mensaje = $"{palabra} no es correcta.\n";
                string[] sugerencias = diccionario.Suggest(palabra).ToArray();
                if (sugerencias.Length > 0)
                    mensaje += $"¿Quisiste decir {string.Join(", ",
sugerencias)}?\n";
            }
            else
                mensaje = $"{palabra} es correcta.\n";
            Console.WriteLine(mensaje);
        } while (palabra != "adios");
    }

}
```

Ejercicio ejemplo para crear nuestras propias librerías:

Con Visual Studio el proceso es bastante sencillo.

1. Primero agregaremos un nuevo proyecto de tipo Biblioteca de clases (.Net Framework), a nuestra solución. Le podemos llamar **MiLibreria.** El espacio de nombres por defecto para empaquetar nuestras clases será el mismo nombre de la librería, en este caso MiLibreria.
   No deberemos tener ningún método Main() pues no se trata de un ejecutable. Aunque podremos cambiarlo desde las propiedades del proyecto. Además, podremos definir otros subespacios de nombres para agrupar nuestras clases en paquetes más específicos. En algún momento alguno de estos subpaquetes podría ir a una nueva librería.

– Vamos añadir una clase de utilidad para ampliar las funcionalidades desde la entrada por la consola. Para ello copia el código siguiente....

```
using System;
namespace MiLibreria
{
    static public class ConsolaAmpliada
    {
        private static void Muestra(string label)
        {
```

```
            if (label != null)
                Console.Write($"{label}: ");
        }
        private static string LeeLinea(bool hidden = true)
        {
            string text = "";
            do
            {
                ConsoleKeyInfo key = Console.ReadKey(true);
                if (key.Key != ConsoleKey.Backspace && key.Key !=
ConsoleKey.Enter)
                {
                    text += key.KeyChar;
                    Console.Write(hidden ? "*" : key.KeyChar.ToString());
                }
                else
                {
                   if (key.Key == ConsoleKey.Backspace && text.Length > 0)
                    {
                        text = text.Substring(0, (text.Length - 1));
                        Console.Write("\b \b");
                    }
                    else if (key.Key == ConsoleKey.Enter)
                    {
                        Console.Write("\n");
                        return text;
                    }
                }
            } while (true);

        }
        public static string LeePassword(bool hidden = true)
        {
            return LeePassword(null, hidden);
        }
        public static string LeePassword(string label, bool hidden = true)
        {
            string passWord;
            bool valid;
            do
            {
                Muestra(label);
                passWord = LeeLinea(hidden);
                valid = passWord.Length > 0;
                if (!valid)
                    Console.WriteLine($"El password debe tener una al
menos un carácter.");
            } while (!valid);
            return passWord;
        }
    }
}
```

La librería deberemos de compilarla para generar el archivo .dll

– A continuación crearemos una aplicación de Consola llamada Ejercici1, a la que añadimos una referencia a la DLL, y en la aplicación haremos una llamada al método LeePAssword, primero usando el nombre completamente calificado

del espacio de nombres que hemos creado. Y después probaremos incluyendo la sentencia using con nuestro espacio de nombres, ***MiLibreria***.

```
using System;
    using MiLibreria;
    public class Program
    {
        static void Main()
        {
            string clave = ConsolaAmpliada.LeePassword("Clave", true);
            Console.WriteLine($"La clave introducida es {clave}");
        }
    }
```

– Modifica los métodos de la librería y comprueba si se han modificado en tu aplicación. Si no se han modificado, ¿que tendrás que hacer para referirte a los nuevos?

2. Otra opción interesante es la de añadir un elemento existente a nuestro proyecto, como por ejemplo cualquiera clase de utilidad. Crea un proyecto nuevo que se llame Mostrar, agrega una clase como la siguiente:

```
static public class Mostrar
{
       private static void Muestra(string label)
       {
          if (label != null) Console.Write($"{label}: ");
        }
}
```

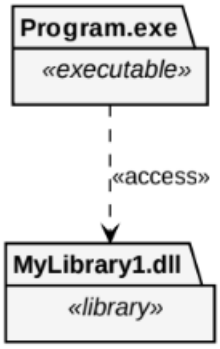
Ahora crea otra aplicación de consola llamada Ejercicio2, agrega la clase **Mostrar,** pero esta vez agregando elemento existente y llama al método Muestra pasándole una frase a Mostrar. ¿Es necesario que la clase sea pública?

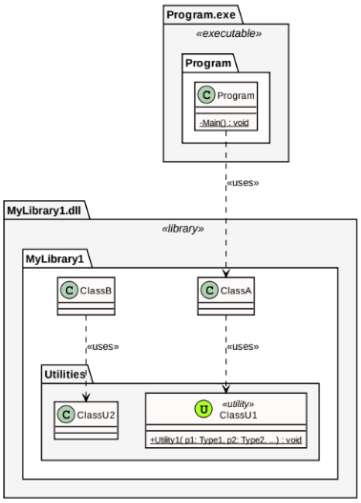
Vuelve al proyecto Mostrar y modifica el método Muestra, añadiendo color al texto de salida, guarda todo el proyecto. Comprueba ejecutando el Ejercicio2,si se muestra el texto de otro color

3. Para terminar vas a comprobar que no es necesario crear DLL para usar las clases que te hayas creado en otros ensamblados. Para ello deberás añadir una referencia, en la aplicación que estamos usando como ejemplo, a cualquier .EXE de los ejercicios de Clases que tengas guardados. Después crea un objeto de esa Clase y procede a usarlo invocando a uno de sus métodos.

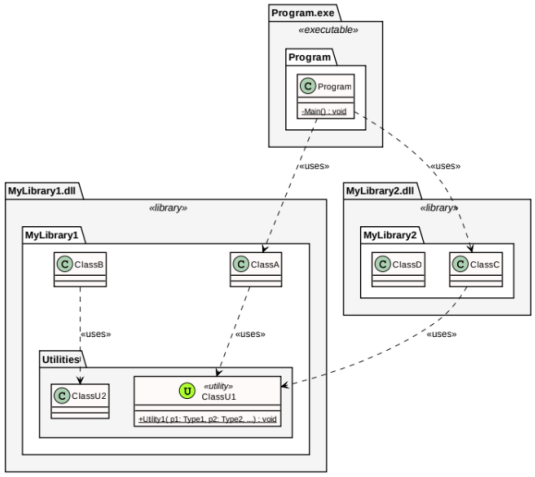
1.3. Gestionar dependencias entre librerías

Vamos a verlo a través de unos ejemplos. Supongamos una librería MyLibrary.dll con el paquete de clases MyLibrary. Dentro de estas la clase ClassA usa un método de utilidad de MyLibrary.Utilities.ClassU1. Las dependencias actuales serán...

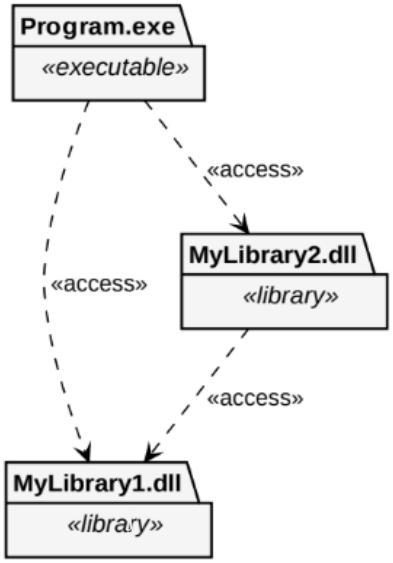


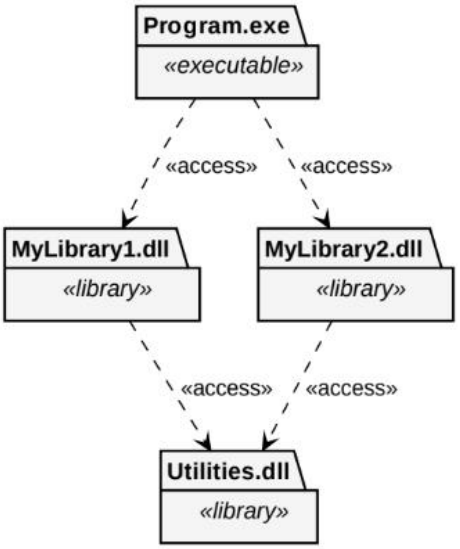


Ahora desde la clase ClassC en una segunda librería MyLibrary2.dll también queremos utilizar el método de utilidad enMyLibrary1.Utilities.ClassU1. Se producirá una situación de doble ámbito de uso, de la clase de utilidad. Pues es usado dentro de la propia MyLibrary1.dll y ahora fuera por parte de una librería diferente.



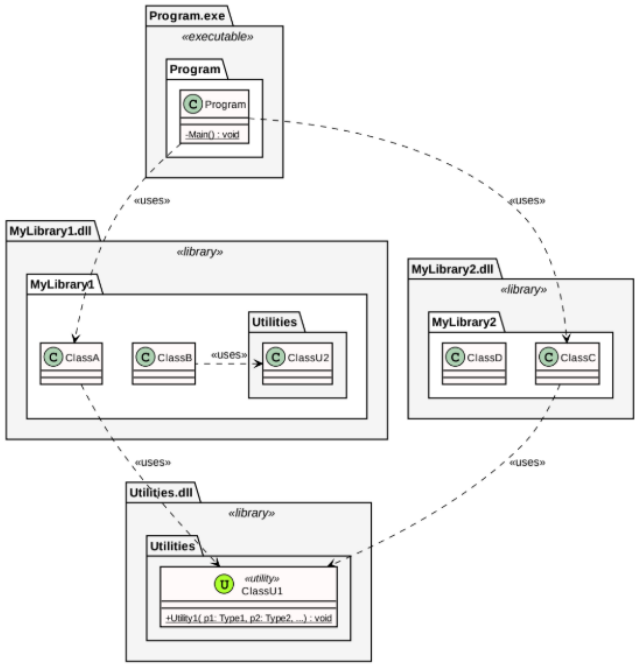
Este doble ámbito, generará una dependencia de MyLibrary1.dll por parte de MyLibrary2.dll cuando realmente, solo hay una clase que queremos reutilizar de ese paquete.
Ahora siempre que queramos usar MyLibrary2.dll vamos a necesitar que esté instalada MyLibrary1.dll cuando hay muchas clases de esta, que no vamos a utilizar y no nos interesan.





La solución será sacar la clase que produce el doble ámbito de uso a una tercera librería, por ejemplo Utilities.dll. Bajando así el acoplamiento elevado que se estaba produciendo por el doble ámbito.

Fijémonos que ahora el doble ámbito de uso ha desaparecido, pues en ambos casos estamos accediendo a una funcionalidad en una dll externa a la nuestra.



2. Definición de propiedades

Una propiedad es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Es la forma de implementar los métodos de acceso (accesores) y

actualización (mutadores), que permiten trabajar de forma segura con los atributos privados en c#.

Su sintaxis es la siguiente:

```
<TipoPropiedad> <NombrePropiedad>
{
   set
   {
       Atributo = value;
   }
   get
   {
       return Atributo;
   }
}
```

set Es un método de actualización y el valor del atributo nos llega en un parámetro especial denominado value. get Es un método de acceso. En una propiedad al menos uno de los dos debe estar definido. Es decir, podremos definir solo uno de los dos, set o get y estarán afectadas por los modificadores de accesibilidad como el resto de métodos.

NOMENCLATURA: Si el nombre del atributo coincide con la propiedad, el atributo irá en camel Casing y puede ir precedido del carácter '_' y la propiedad toma el nombre del atributo en Pascal Casing.

Ejemplo:
Para acceder o cambiar el atributo Fila de nuestra clase PuntoConsola con una propiedad, en primer lugar cambiaremos el nombre del atributo Fila por fila o por _fila. Después añadiremos la siguiente propiedad:

```
public ushort Fila
{
  set { if(value <= 24) fila = value;}
  get { return fila;  }
}
```

Si en el Main() ponemos:

```
Console.WriteLine(punto.Fila); // Veremos el valor de la fila.
Internamente se llama al get
punto.Fila = 12; // Cambiaremos el valor de la fila a 12.  Internamente
se llama al set
```

En C# 3.0 y versiones posteriores, aparecen las **propiedades autoimplementadas** que hacen que la declaración de propiedad sea más concisa cuando no se requiere ninguna lógica adicional en los descriptores de acceso de la propiedad. Al declarar una propiedad tal y como se muestra en el ejemplo siguiente, el compilador crea un campo de respaldo privado y anónimo al que solamente puede obtenerse acceso a través de los descriptores de acceso get y set de la propiedad.

```
class Trabajo
{
      // Propiedades autoimplementables
```

```
        public double SueldoTotal { get; set; }
        public string Nombre { get; set; }
        public int Identificador { get; set; }

        // Constructor
        public Trabajo(double sueldo, string nombre, int id)
        {
           this.SueldoTotal= sueldo;
           Nombre = nombre;
           this.Identificador = id;
        }
   }

   class Program
   {

        static void Main()
        {
            Trabajo trabajador = new Trabajo(4987.63, "Alicante",
                                             90108);
            trabajador.SueldoTotal += 499.99;
        }
    }
```

3. Redefinición de operadores

En temas anteriores ya hemos vimos los operadores unarios, binarios, de cast, etc… usados con tipos simples. Para dos enteros la suma binaria + los sumaba 2 + 3 = 5 y para el caso del tipo cadena esta suma significaba concatenación por ejemplo "Hola " + "Caracola" = "Hola Caracola".

En los lenguajes orientados a objetos podremos darle el significado que queramos a cualquiera de los operadores existentes cuando lo apliquemos a dos objetos de una clase definida por nosotros (siempre y cuando la operación tenga sentido).

Los operadores aritméticos, lógicos y de comparación son redefinibles pero, no todos los operadores se pueden redefinir como (new, =). Además, algunos como [ ] no lo son con esta sintaxis.

La forma en que se redefine un operador depende del tipo de operador del que se trate, ya que no es lo mismo definir un operador unario que uno binario. Sin embargo, como regla general podemos considerar que se crean definiendo un método público y estático cuyo nombre sea el símbolo del operador a redefinir y venga precedido de la palabra reservada operator. Es decir, se sigue una sintaxis de la forma:

```
public static <tipoDevuelto> operator <símbolo>(<operandos>)
{
   <cuerpo del método>
}
```

Los modificadores public y static pueden permutarse si se desea, lo que es importante es que siempre aparezcan en toda redefinición de operador. Se puede redefinir tanto operadores unarios como binarios, y en <operandos> se ha de incluir tantos parámetros como operandos pueda tomar el operador a redefinir, ya que cada uno representará a uno de sus operandos. Por último, en <cuerpo> se ha de escribir las instrucciones a ejecutar cada vez que se aplique la operación, como toda operación tiene un resultado el operador ha de devolver algo.

Ejemplo de redefinición del operador + para sumar números complejos:

```
class Complejo
{
    private double parteReal;
    private double parteImaginaria;
    public Complejo(double parteReal, double parteImaginaria)
    {
        this.parteReal = parteReal;
        this.parteImaginaria = parteImaginaria;
    }
    public override string ToString()
    {
        string texto = $"{parteReal:G}";
        texto += (parteImaginaria > 0D) ? " + " : " - ";
        texto += $"{Math.Abs(parteImaginaria):G}i";
        return texto;
    }
    public static Complejo operator +(Complejo op1, Complejo op2)
    {
        double parteReal =  op1.parteReal + op2.parteReal;
        double parteImaginaria = op1.parteImaginaria + op2.parteImaginaria;
        return new Complejo(parteReal, parteImaginaria);
    }
}
```

Uso de el operador:

```
Complejo c1 = new Complejo(3, 2);
Complejo c2 = new Complejo(5, 2);
Complejo c3 = c1 + c2;
Console.WriteLine(c3); // Mostrará 8 + 4i
```

Operador Cast → Se puede tomar como un caso específico la redefinición de transformación de tipos, ya que no sigue exactamente la misma sintaxis que los anteriores operadores. Para este operador en concretos, nos podemos encontrar con dos casos: el Cast explícito, y el implícito menos usado.

Cast explicit
public static explicit operator <tipo a convertir>(<operando>)
{
<cuerpo del método>
}

```
public static explicit operator float(Complejo c)
{
    return Convert.ToSingle(c.parteReal);
}
```

```
Complejo c = new Complejo(3.7, 2.4);
float f = (float)c; // Asignará 3.7 a f
double d = (double)c; // Daría error porque no está definido
                      // el operador de cast explícito a double.
```

Cast implicit

```
public static implicit operator <tipo a convertir>(<operando>)
{
   <cuerpo del método>
}
```

```
public static implicit operator double(Complejo c)
{
    return Convert.ToSingle(c.parteReal);
}
```

```
Complejo c = new Complejo(3.7, 2.4);
double d = c; // Asignará 3.7 a d
```

4. Indizadores

Un indizador es una definición de cómo se puede aplicar el operador de acceso a tablas ([ ]) a los objetos de un tipo de dato. Esto es especialmente útil para hacer más clara la sintaxis de acceso a elementos de objetos que puedan contener colecciones de elementos, pues permite tratarlos como si fuesen tablas normales. Es un concepto muy parecido al de las propiedades.

Para indexar no necesariamente utilizaremos un entero, también podremos utilizar otros tipos como cadenas, tipos enumerados, etc, también se pueden definir varios indizadores en un mismo tipo siempre y cuando cada uno tome un número o tipo de índices diferente.

```
<tipoIndizador> this[<indices>]
{
    set
    {
       <códigoEscritura>
    }
    get
    {
       <códigoLectura>
    }
}
```

Ejemplo:

```
namespace ConsoleApplication1
{
    class P1
    {
        private string[] dato;

        public P1()
        {
            this.dato = new string[100];

        }
        public string this[int indice]
        {
            get
            {
                if (indice < dato.Length) return dato[indice];
                else return "";
            }
            set
            {
                if (indice < dato.Length) dato[indice] = value;
            }
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            P1 p = new P1();

            p[1] = "hola";
            p[10] = "Prueba";
            Console.WriteLine("p[1] vale {0} \n p[10] vale {1}", p[1],
                                                                 p[10]);
        }
    }
}
```

Si dato fuese public la nomenclatura para acceder a sus elementos sería p.dato[1] = “juan”. Como puede verse no es tan clara como cuando se utilizan indizadores.

Puede parecer no muy útil el uso de indizadores, después de ver el anterior ejemplo, pero vamos a suponer otro caso en el que no se aplique a un array, sino que sea un indizador que nos devolverá una salida a partir de un enumerado de entrada.

```
enum Dias { lunes, martes, miercoles, jueves}
class Horario
{
    public String this[Dias d]
    {
```

```
        get
        {
            switch (d)
            {
                case Dias.lunes: case Dias.miercoles:
                    return "17:00";
                case Dias.martes:case Dias.jueves:
                    return "17:30";
            }
            throw new IndexOutOfRangeException("Este día no consta de
entrenamiento");
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Console.WriteLine(new Horario()[(Dias)4]);
        }catch(IndexOutOfRangeException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

5. Interfaces

Una interfaz es la definición de un conjunto de métodos para los que no se da implementación, sino que se les define de manera similar a como se definen los métodos abstractos. Es muy parecido a definir una clase abstracta pura, pero sin ningún tipo de atributo, constructor ni modificador de acceso (public, private, etc…). Como en las clases abstractas, las interfaces son tipos referencia, no puede crearse objetos de ellas sino sólo de tipos que deriven de ellas, y participan del polimorfismo. Las interfaces pueden estar compuestas de métodos, propiedades, eventos, indizadores o cualquier combinación de estos cuatro tipos de miembros. Una interfaz no puede contener campos. Los miembros de interfaz son automáticamente públicos. Para aplicar un interfaz a una clase o una estructura haremos que esta herede del interfaz con la sintaxis de herencia que hemos usado hasta ahora.

```
<modificadores> interface I<identificador>:<interfacesBase>
{
   <interfaces de métodos o propiedades>
}
```

Pueden implementarse en la mayoría de lenguajes con idénticas características:

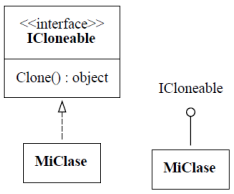
- Es posible la herencia múltiple de interfaces.
- Un interfaz puede heredar de otro interfaz.
- Si una clase hereda de un interfaz esta deberá implementar todos los métodos definidos en la misma.

Nota: El identificador siempre irá precedido por la letra I para distinguirlo de otro tipo de clases.

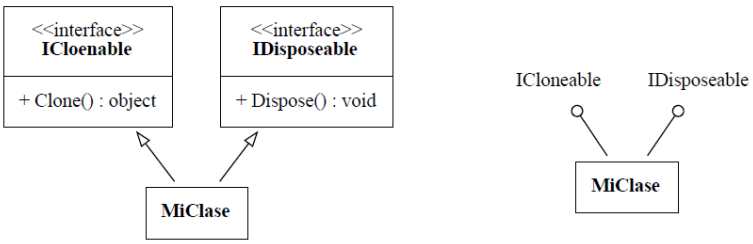
Para implementar un miembro de interfaz, el miembro correspondiente de la clase debe ser público, no estático y tener el mismo nombre y la misma firma que el miembro de interfaz.

Interfaces en los diagramas de clases uml

Para expresar que la clase MiClase implementa el interfaz Icloneable.
Podremos expresarlo de las forma siguiente →
MiClase ahora está obligada a implementar el método publico Clone con idéntica signatura.



Podremos Hacer Que Una Clase Implemente O “herede” De Más De Una Interfaz



Ejemplo:

```
using System;
interface IMedia
{
    void Play();
    void Stop();
    void Pause();
    float Duracion { get; }
}


class Video : IMedia
{
    private float Segundos;
    public Video(float segundos)
    {
        this.Segundos = segundos;
    }
    public void Play()
    {
        Console.WriteLine("Ha pulsado usted el encendido");
```

```
    }
    public void Stop()
    {
        Console.WriteLine("Ha pulsado usted la parada");
    }
    public void Pause()
    {
        Console.WriteLine("Ha pulsado usted la pausa");
    }
    public float Duracion
    {
        get{return Segundos;}
        set { Segundos = value; }
    }
}
class Program
{
    public static void Main()
    {
        Video miVideo = new Video(123.3F);
        miVideo.Play();
        miVideo.Duracion = DateTime.Now.Second;
        Console.WriteLine("Para seguir pulse  cuarquier tecla");
        Console.ReadKey();
        miVideo.Stop();
        Console.WriteLine("La duración total ha sido {0}",
                          DateTime.Now.Second-miVideo.Duracion);
  }
}
```

Existen muchos interfaces ya definidos en las BCL bastante utilizados. De entre ellos podremos destacar los siguientes:

- IDisposable -> Me indicará que el objeto debe implementar el método void Dispose() que se encargará de liberar los recursos usados por el objeto.

- IClonable -> Me indicará que puedo crear copias del objeto, puesto que me obliga a implementar un “constructor copia” con el interfaz Object Clone()

- IComparable ->Me indicará que el objeto debe implementar el método int CompareTo(Object otro) que me servirá para comparar dos objetos de la misma clase.

- IEnumerable -> Esta interfaz se implementa para colecciones, y permite recorrerlas. La veremos posteriormente, cuando hayamos introducido las colecciones.

Podríamos utilizar interfaces nuestros para hacer lo mismo, pero perderíamos interoperabilidad con el resto de clases de las BCL.

```
class MiClase : ICloneable, IComparable
{
    int Algo;

    private MiClase(MiClase c)
    {
```

```
            Algo = c.Algo;
        }
        protected virtual object clone()
        {
            return new MiClase(this);
        }
        public object Clone()
        {
            return clone();
        }
        public int CompareTo(object obj)
        {
             if (obj == null) return 1;
             else
            {
                MiClase mc = obj as MiClase;
                if (mc == null) throw new ArgumentException(
                "El objeto a comparar no es de MiClase");
                if (Algo == mc.Algo) return 0;
                else return (Algo > mc.Algo) ? 1 : -1;
            }

        }
}
```

5.1. Instrucción using

Se utiliza para instanciar objetos que contiene recursos no gestionados, esto es, que no son liberados por el recolector de basura.

Un recurso es una clase o estructura que implementa **System.IDisposable**, que incluye un único método sin parámetros denominado **Dispose**. El código que utiliza un recurso puede llamar a **Dispose** para indicar que ya no lo necesita.

La sentencia Using garantiza que se llama a Dispose aunque se produzca una excepción. Dentro del bloque using, el objeto es de solo lectura y no se puede modificar ni reasignar puesto que dejaría de tener una referencia y no se liberaría.

La instrucción **using** obtiene uno o varios recursos, ejecuta una instrucción y a continuación elimina el recurso. La sintaxis es la siguiente:

```
using (TipoIDisposabe r1 = e1, r2 = e2, ..., rN = eN)
{
    // Ámbito de uso de solo lectura de r1, r2, … , rN
}
```

Las **instrucciones using** pueden anidarse:

```
using (TipoIDisposabe r1 = e1)
{
   using (TipoIDisposabe r2 = e3) {
  // También son anidables
  }
}
```

Las variables locales que se declaran en la *sentencia using* son de sólo lectura y deben incluir un inicializador. Se producirá un error en tiempo de compilación si la instrucción incrustada intenta modificar estas variables locales (por medio de asignación o utilizando los operadores ++ y --) o bien si las pasa como parámetros ref o out.

Una instrucción **using** se traduce en tres partes: adquisición, uso y eliminación. El uso de un recurso está incluido implícitamente en una instrucción try que contiene una cláusula finally. La cláusula finally elimina el recurso. Si se obtiene un recurso null no se llama a Dispose y no se inicia ninguna excepción.

Una instrucción using de la forma

```
using (ResourceType resource = expression) statement
```

Corresponde a una de dos posibles expansiones (según referencia o valor)

```
ResourceType resource = expression;
try
{
    statement;
}
finally
{
   if (resource != null)
     ((IDisposable)resource).Dispose(); //En Caso de recurso
referencia
//((IDisposable)resource).Dispose(); //En Caso de recurso valor
}
```

En el ejemplo siguiente se crea un archivo llamado log.txt y se escriben dos líneas de texto en él. Posteriormente, se abre el mismo archivo para leerlo y copiar las líneas de texto que contiene en la consola.

```
using (TextWriter w = File.CreateText("log.txt"))
{
  w.WriteLine("Línea Uno");
  w.WriteLine("Línea Dos");
}
using (TextReader r = File.OpenText("log.txt"))
{
  string s;
  while ((s = r.ReadLine()) != null) Console.WriteLine(s);
 }
```

Dado que las clases TextWriter y TextReader implementan la interfaz IDisposable, en el ejemplo se pueden utilizar instrucciones using para garantizar que el archivo subyacente esté correctamente cerrado después de las operaciones de lectura o escritura.

6. Genéricos

C# 2.0 incluye esta característica que permite definir los tipos dentro de una clase de forma parametrizada al instanciar un objeto de la misma. De tal manera que para objetos diferentes estos tipos pueden cambiar. (Polimorfismo paramétrico). Expresaremos el tipo o los tipos genéricos a través de una o más letras mayúsculas usadas a lo largo de la definición de la clase. Aunque se suele usar la letra T, podremos usar cualquier otra que nos indique que representa el tipo parametrizado. A estas letras se les denomina PARAMETROS TIPO. Podremos usarlos en clases, métodos, interfaces, delegados, etc…

Los genéricos nos evitarán problemas en tiempo de ejecución derivados de un mal uso del tipo Object, aclaran y reducen la cantidad de código para quien usa nuestras clases.

Su definición es:

```
<modificadores> class <identificador><parámetros tipo separados comas>
{
      <cuerpo de la clase>
}
```

Para instanciar un objeto de una clase genérica supongamos A de tipo entero:

```
A<int> objA = new A<int>();
```

```
public class A<T>
{
   private T dato;
   public void EstablecerValor(T dato)
   {
     this.dato = dato;
   }
}
public class AAA<C, V>
{
   private C clave;
   private V valor;
   public V DameValor(C clave)
   {
      return valor;
   }
}
class Program
{
    public static void Main()
    {
        A<int> x=new A<int>();
        x.EstablecerValor(5);
    }
}
```

Lo que hace el compilador en tiempo de ejecución será lo siguiente, transformará todas las instancias de <T> en int, quedando la clase A

```
public class A
{
   private int dato;
   public void EstablecerValor(int dato)
   {
      this.dato = dato;
   }
}
```

6.1. Métodos genéricos

C# permite definir no sólo tipos genéricos, sino que también puede hacerse genéricos a métodos individuales, tanto de instancia como estáticos, sin necesidad de que lo sea la clase o estructura en la que el método está definido. Es importante señalar que un método de un tipo genérico no es genérico simplemente por eso, para que un método genérico sea considerado así debe tener al menos un placeholder de tipo.

La sintaxis es casi la misma que al declarar un método tradicional, salvo que de nuevo aparecen los placeholders de tipos entre < >, por ejemplo, en el siguiente método se declara únicamente T:

```
static void WriteType<T>()
{
    var a = typeof(T);
    Console.WriteLine(a.Name);
}
```

```
WriteType<int>(); // Int32
WriteType<Program>(); // Program
```

Al igual que con los tipos, una vez indicado el placeholder podemos utilizarlo tanto en la firma del método como en el cuerpo del mismo:

```
static T GetDefault<T>(T useless)
{
    return default(T);
}
```

Para usarlos tenemos dos opciones:

1. Indicar explícitamente los tipos a la hora de realizar la llamada
2. Permitir que el compilador infiera a qué tipos nos referimos a partir de los argumentos pasados

```
var dflt = GetDefault<int>(10);

var dflt2 = GetDefault(4.0); // Se infiere que es double a partir del
argumento pasado al método

// int dflt3 = GetDefault(DateTime.Now); // Error, DateTime no se puede
convertir a int implícitamente
```

Supongamos el siguiente ejemplo:

```
static void Swap<T>(ref T a, ref T b)
{
    T t = a;
    a = b;
    b = t;
}
```

Tendríamos un método que podríamos usar con cualquier tipo que deseemos:

```
int uno = 1, dos = 2;
Swap<int>(ref uno, ref dos);
Swap(ref uno, ref dos); // Se infiere int
float d1 = 10, d2 = 20;
Swap(ref d1, ref d2); // Se infiere float
```

Cuando usamos métodos genéricos es posible **incurrir en errores**, los más comunes son que el compilador no pueda inferir los tipos a los que nos referimos, como en este caso que estamos usando **Swap** con un **int** y un **float**:

```
Swap(uno, d2); // Error, los tipos no pueden ser inferidos
```

o que intentemos usar un tipo de datos que no corresponde si especificamos explicitamente los tipos.

```
Swap<int>(ref uno, ref d2); // Error, d2 no es entero
```

Restricciones

Al igual que los tipos genéricos, los métodos también aceptan restricciones usando la palabra reservada `where`, por ejemplo, en el método siguiente se restringe que tanto el tipo `T` como `W` sean `Athlete` o derivados de este. Además de que `T` tenga un constructor público sin parámetros:

```
static T ChangeSport<T, W>(W toCast) where T : Athlete, new()where W : Athlete
{
    // ...
```

6.2. Utilización de genéricos en lugar del tipo Object

Supongamos una clase Fecha que hereda del interfaz IComparable, este nos obliga a implementar el método int CompareTo(Object o) donde en tiempo de compilación no se especifica que el **objeto** o tiene que ser de tipo fecha. Esto significa que podremos pasar cualquier objeto pues todos heredan de Object. Obligándonos a un código de control para evitar este problema en tiempo de ejecución. Además del obligatorio cast a fecha.

```
class Fecha : IComparable
{
        int dia;
        int mes;
        int año;
        public int CompareTo(Object objFecha)
```

```
        {
            if (objFecha is Fecha == false)
            throw new Exception("Aviso");
            Fecha fecha = (Fecha)objFecha;
            if (fecha.año > año) return -1;
            else if (fecha.año < año) return 1;
            else if (fecha.mes < mes) return -1;
            else if (fecha.mes > mes) return 1;
            else if (fecha.dia < dia) return -1;
            else if (fecha.dia > dia) return 1;
            else return 0;
        }
 }
```

Sin embargo si nosotros utilizamos la definición genérica de IComparable no necesitaremos hacer el control pues en tiempo de ejecución se creará la definición del interfaz con el método int CompareTo(Fecha fecha) Sin embargo en tiempo de compilación ya nos avisará si no estamos pasando un objeto de tipo Fecha, ahorrandonos además el tan tedioso cast al tipo Fecha.

```
class Fecha : IComparable<Fecha>
{
        private int dia;
        private int mes;
        private int año;

        public int CompareTo(Fecha fecha)
        {
                if (fecha.año > año)  return -1;
                else if (fecha.año < año) return 1;
                else if (fecha.mes < mes) return -1;
                else if (fecha.mes > mes) return 1;
                else if (fecha.dia < dia) return -1;
                else if (fecha.dia > dia) return 1;
                else return 0;
        }
}
```

EXPECIFICACIONES

¿Cómo inicializar un dato genérico a su valor por defecto? Usaremos la expresión default(T)

```
class A<T>
{
  T dato = default(T);
}
```

No podremos usar los parámetros tipo (T, U, K, etc..) como nombre de clases base, interfaces, atributos.

Deberemos llevar cuidado con el polimorfismo funcional.

```
class A<T>
{
```

public void Funcion(int p1, string p2) { ;}
public void Funcion(T p1, string p2) { ;}
}

Esta declaración es correcta y tendriamos dos signaturas si declaramos A<string> obj = new A<string>(); pero ¿Qué pasa si declaramos A<int> obj = new A<int>();? Tendremos dos signaturas iguales y se ejecuta la no genérica.
A un objeto declarado a partir de un parámetro tipo:

- **Sólo podremos aplicarle operaciones como si fuera de tipo Object y no podremos usar operadores.**
- **No podremos realizarle un cast explícito sin pasar previamente a object.**

Para ver más claramente lo anterior supongamos este código:

```
class Fecha <T>
{
   T dia;
   T mes;
   T año;

   public Fecha(T dia, T mes, T año)
   {
       this.dia = dia;
       this.mes = mes;
       this.año = año;
   }
   public Fecha(Fecha<T> f)
   {
       this.dia = f.dia;
       this.mes = f.mes;
       this.año = f.año;
    }
 public int CompareTo<T>(Fecha<T> fecha)
 {
  if (Convert.ToInt16(fecha.año) >Convert.ToInt16( año))  return -1;
  else if(Convert.ToInt16(fecha.año) < Convert.ToInt16(año)) return 1;
  else if(Convert.ToInt16(fecha.mes) < Convert.ToInt16(mes)) return -1;
  else if(Convert.ToInt16(fecha.mes) > Convert.ToInt16(mes)) return 1;
  else if(Convert.ToInt16(fecha.dia) < Convert.ToInt16(dia)) return -1;
  else if(Convert.ToInt16(fecha.dia) > Convert.ToInt16(dia)) return 1;
       else return 0;
 }
}
 //Clase con el programa principal
 class Program
 {
    static void Main(string[] args)
    {
       Fecha<int> f = new Fecha<int>(1,1,1);
       Fecha<int> j=new Fecha<int>(2,2,2);
       Console.WriteLine(j.CompareTo(f));
    }
 }
```

6.3.- Definición de Restricciones

Podremos definir restricciones de tipo para los parámetros tipo. Se especifican con la siguiente cláusula asociada a la definición del tipo genérico.

```
<parametroTipo> where parametroTipo:restricción
```

```
class A<T> where T: struct
{
// T sólo podran ser tipos valor.
}
```

Definición de Restricciones

Tendremos diferentes tipos de restricciones: Restricción	Objetivo
De herencia	El tipo debe heredar de una clase base determinada. <T> where T: ClaseBaseDeT
De interfaz	El tipo debe implementar una interfaz determinada. <T> where T: IinterfazAImplementar
De tipo referencia	El tipo debe ser referencia <T> where T: class
De tipo valor	El tipo debe ser valor <T> where T: struct
De constructor	El tipo debe tener un constructor sin parámetros. <T> where T: new

7. Principios S.O.L.I.D

La experiencia en el desarrolló usando el paradigma POO, dio lugar a cinco reglas que todo desarrollador OO debería seguir para crear un sistema que sea escalable y fácil de mantener a través del tiempo.
Estas cinco reglas son conocidas como los principios SOLID y las vamos a enumerar a continuación.
Algunas ya las hemos visto indirectamente durante el curso.

7.1. Principio de Responsabilidad Única (SRP)

Descrito por Robert C. Martin. Este principio establece que un componente del software (método, clase o módulo) debe estar centrado en una única tarea (tener solo una responsabilidad).
Cómo ocurría en programación modular, que si un módulo hacía más de una cosa genera acoplamiento. Lo mismo sucederá con nuestras clases.
No siempre es tan obvio su incumplimiento. Por eso, siempre que añadamos un método a una clase deberemos pensar si realmente debería ir en la misma o realmente se está convirtiendo en un “Cajón Desastre”.

Veamos un ejemplo de una clase Rectangulo:

```
public class Rectangulo
{
    public double Alto;
    public double Ancho;
    public static double SumaAreas(Rectangle[] rectangles)
    {
        // Código necesarios
        return 0;
    }
    public static double SumaPerímetro(Rectangle[] rectangles)
    {
        // Código necesarios
        return 0;
    }
}
```

Mientras que el programa principal se encarga de crear los rectángulos y llamar a los métodos respectivos para obtener las sumas:

```
static void Main(string[] args)
{
   Rectangle[] rectangulos = new Rectangle[]{
                             new Rectangle {Ancho = 10, Alto = 5},
                             new Rectangle {Ancho = 4, Alto = 6},
                             new Rectangle {Ancho = 5, Alto = 1},
                             new Rectangle {Ancho = 8, Alto = 9}};
   double sumaDeAreas = Rectangle.SumaAreas(rectangulos);
   double sumaDePerimetros = Rectangle.SumaPerímetro(rectangulos);
}
```

La violación ocurre al momento de declarar los métodos SumaAreas y SumaPerimetros dentro de la misma clase que Rectangulo, y es que a pesar de que están relacionadas con el rectángulo como tal, la sumatoria forma parte de nuestra lógica de la aplicación, no de la lógica que podría tener un rectángulo en la vida real.

Para cumplir con el principio, quitamos la funcionalidad de sumatorias de la clase Rectangulo e introducimos un par de clases encargadas de realizar las operaciones sobre el los conjuntos de rectángulos, su código es más o menos este:

```
public class CalculosSobreAreas
{
   public static double Suma(Rectangle[] rectangles)
   {
            //
   }
}
public class CalculoSobrePerimetros
{
   public static double Suma(Rectangle[] rectangles)
   {
            //
   }
}
```

Entonces así cada clase tiene una sola responsabilidad: una representa un rectángulo y las otras se encargan de hacer operaciones relacionadas con ellos.

7.2 Principio de Abierto/Cerrado (OCP)

Establece que el diseño debe ser abierto para poderse extender, pero cerrado para poderse modificar. En otras palabras, el Software debe ser diseñado pensando en el crecimiento de la aplicación, pero el nuevo código debe requerir el menor número de cambios en el código existente. Abierto a lo nuevo, cerrado para lo viejo.

Formas de aplicarlo en POO:

- El uso más común de extensión es mediante el mecanismo de herencia y la invalidación o sobreescritura de métodos.
- Utilizando abstracción, por ejemplo utilizando métodos que acepten una interface, de manera que podemos pasar cualquier clase que lo implemente.
- Usando el Patrón Decorator por ejemplo FileStream y su "decorator" BufferedStream.

Cómo la herencia y abstracción ya las hemos tratado, veamos un ejemplo de uso de OCP aplicando el patrón Decorator. Supongamos que realizamos la siguiente abstracción de ordenador para calcular el coste de diferentes configuraciones de ordenadores.

```
public abstract class Ordenador
{
    public abstract double Precio { get; }
}
public class OrdenadorBasico : Ordenador
{
    private const double PROCESADOR = 56D;
    private const double HDD = 30D;
    private const double GRAFICA = 41.99D;
    private const double RAM = 23.5D;

    public override double Precio
    {
       get { return PROCESADOR + HDD + GRAFICA + RAM; }
    }
}
```

Podríamos extender nuevas configuraciones como OrdenadorGammer, OrdenadorDeOficina y así todas las que se nos ocurran, que pueden ser muchísimas. **Una forma alternativa más "flexible" sería usar el Patrón Decorator.**

Usando la abstracción Ordenador (podría ser también un Interface). Definiremos el patrón decorador que extenderá Ordenador y que referenciará a la configuración del ordenador actual (acumulada) y deberá implementar la propiedad Precio donde añada a esta configuración el precio de dicho componente.

```
public abstract class ComponenteDecorator : Ordenador
{
   protected Ordenador ConfiguracionActual { get; set; }
   public override abstract double Precio { get; }
}
```

Definiremos ahora extensiones de Ordenador que envolverán el ordenador con la configuración hasta el momento y la "decorarán" añadiéndole el precio de un componente.

```
class SSDRapido : ComponenteDecorator
{
        public SSDRapido(Ordenador configuracion)
        {
            ConfiguracionActual = configuracion;
        }
        public override double Precio
        {
            get { return ConfiguracionActual.Precio + 255.20D; }
        }
}
public class Procesador8Nucleos : ComponenteDecorator
{
        public Procesador8Nucleos(Ordenador configuracion)
        {
            ConfiguracionActual = configuracion;
        }
        public override double Precio
        {
            get { return ConfiguracionActual.Precio + 360D; }
        }
}
```

De forma análoga añadir todos los componentes que queramos a posteriori sin modificar el código anterior.
Definimos una clase ordenador con una configuración básica o vacía que iremos "decorando"

```
public class OrdenadorVacio : Ordenador
{
  public override double Precio
  {  get { return 0D; } }
}
```

Por último, podremos definir la configuración del ordenador de la siguiente manera...

```
static void Main()
{
   Ordenador gammer = new OrdenadorVacio();
   gammer = new SSDRapido(gammer);
   gammer = new Procesador8Nucleos(gammer);
   gammer = new ModuloRAM16GB(gammer);
   // …
   Console.WriteLine($"Precio: {gammer.Precio}");
}
```

7.3 Principio de Sustitución de Liskov (LSP)

Ya lo hemos usado y básicamente se trata de la importancia de definir bien las subclases para que también puedan ser tratadas como la propia abstracción. Básicamente lo que nos dice este principio es que 'Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. En otras palabras, podríamos decir que después de la sustitución, no debería ser necesario ningún otro cambio para que el programa continúe funcionando como lo hacía originalmente.

Veamos un par de ejemplos ilustrativo que nos generarán problemas con el LSP.

Suponemos las siguientes clases con su relación de herencia:

```
class Vehiculo
{
   public string Marca { get; set; }
   public string Modelo { get; set; }
   public int Cilindrada { get; set; }
}
class Ciclomotor : Vehiculo
{
   public string ObtenerNumLicencia(){}
}
class Coche : Vehiculo
{
   public string ObtenerMatricula(){}
}
class Impuestos
{
   public void CalcularImpuesto(Vehiculo vehiculo)
   {
      string matricula = ((Coche)vehiculo).ObtenerMatricula();
      ServicioCalculoImpuestos(matricula, vehiculo.Cilindrada);
    }
}
```

LSP afirma que si tenemos dos objetos de tipos diferentes –Coche y Ciclomotor– que derivan de una misma clase base –Vehículo–, deberíamos poder reemplazar cada uno de los tipos –Coche/Ciclomotor y viceversa– allí dónde el tipo base –Vehículo– esté implementado. En el ejemplo anterior tenemos un claro caso de violación del LSP, ya que la ejecución del método CalcularImpuesto generará una excepción de conversión de tipo si el objeto pasado por parámetro es de tipo Ciclomotor en lugar de Coche, pese a que ambas clases derivan de la misma clase base Vehículo. Podríamos modificar el código para evitar la excepción, pero no sería una solución correcta.

Una posible solución podría ser la siguiente:

```
abstract class Vehiculo
    {
        public string Marca { get; set; }
        public string Modelo { get; set; }
        public int Cilindrada { get; set; }
        public abstract string Informacion();
    }
    class Ciclomotor : Vehiculo
    {
        public override string Informacion()
        {
            return "Número Licencia";
        }
    }

    class Coche : Vehiculo
    {
        public override string Informacion()
        {
            return "Número Matrícula";
        }
    }
    static class Informadora
    {
        public static string Información(Vehiculo vehiculo)
        {
            return vehiculo.Informacion();
        }
    }

    class Program
    {
        static void CalcularImpuesto(Vehiculo vehiculo)
        {
            Console.WriteLine(Informadora.Información(vehiculo));
        }

        static void Main(string[] args)
        {
            CalcularImpuesto(new Coche());
            CalcularImpuesto(new Ciclomotor());
        }
    }
```

Otro ejemplo que no respeta el principio de sustitución de Liskov podría ser el siguiente. Aunque en la vida real tenemos “claro” que un cuadrado es un rectángulo con los dos lados iguales. Que pasa cuando implementamos esta premisa con código.

```
class Rectangulo
{
    public virtual int Ancho { get; set; }
    public virtual int Alto { get; set; }

    public int Area
    {
        get { return Ancho * Alto; }
    }
}
```

Invalidamos las propiedades para obligar que sea un cuadrado.

```
class Cuadrado : Rectangulo
{
    public override int Ancho
    {
        set { base.Ancho = value; base.Alto = value; }
    }
    public override int Alto
    {
        set { base.Ancho = value; base.Alto = value; }
    }
}
```

Ahora si pasamos el siguiente test...

```
private static void TestArea(Rectangulo r)
{
    r.Ancho = 5;
    r.Alto = 4;
   Console.WriteLine(r.Area==20?"El area es 20":$"Área {r.Area} y no 20");
 }
```

Vemos como el LSP no se cumple en el segundo Test, ya que al asignar 4 a Alto el Ancho pasa a ser también 4 para cumplir la restricción de cuadrado y el área me devolverá 16.

```
static void Main(string[] args)
{
    TestArea(new Rectangulo());
    TestArea(new Cuadrado());

}
```

```
El area es 20
Área 16 y no 20
Presione una tecla para continuar . . .
```

En nuestro ejemplo el Cuadrado no puede reemplazar al Rectángulo ya que el comportamiento de nuestro test debería ser modificado para que siga funcionando correctamente. De aquí se deduce que el Cuadrado puede ser una sub-clase del Rectángulo, pero no es un sub-tipo, según la definición de Liskov. Por lo tanto, que una clase herede de otra no nos asegura el principio LSP.

Hay varias posibilidades en función del caso en el que nos encontremos. Lo más habitual será ampliar esa jerarquía de clases. Podemos extraer a otra clase padre las características comunes y hacer que la antigua clase padre y su hija hereden de ella. Al final lo más probable es que la clase tenga tan poco código que acabes teniendo un simple interfaz. Esto no supone ningún problema en absoluto:

```
interface IRectangulo
{
   int Ancho { get; }
        int Alto { get; }

        int Area { get; }
    }
   class Rectangulo:IRectangulo
   {
        ////Implementación
    }

    class Cuadrado : IRectangulo
    {
       ///Implementación
    }
```

Pero para este caso en particular, nos encontramos con una solución mucho más sencilla. La razón por la que no se cumple que un cuadrado sea un rectángulo, es porque estamos dando la opción de modificar el ancho y alto después de la creación del objeto. Podemos solventar esta situación simplemente usando **inmutabilidad**.
La inmutabilidad consiste en que **una vez que se ha creado un objeto, el estado del mismo no puede volver a modificarse**. La inmutabilidad tiene múltiples ventajas, entre ellas un mejor uso de memoria (todo su estado es final) o seguridad en múltiples hilos de ejecución.

```
class Rectangulo
{
   readonly int Ancho;
   readonly int Alto;
   public Rectangulo(int ancho, int alto)
   {
       Ancho = ancho;
       Alto = alto;
   }
   public int Area
   {
```

```
        get { return Ancho * Alto; }
    }
}
class Cuadrado : Rectangulo
{
    public Cuadrado(int ancho, int alto):base(ancho, alto){}
}
```

7.4 Principio de Segregación de Interfaces (ISP)

Este principio viene a decir, que ninguna clase debería implementar métodos definidos en un interface, que luego no va a usar o no tienen sentido. Evitaremos pues, interfaces grandes (fat interfaces) que definan muchos métodos. Por tanto, las reduciremos a aquellos que siempre se van a dar claramente en todas las clases que la implementen.
También podremos usar el mecanismo de extensión o herencia de interfaces para realizar dicha segregación.

Supongamos la siguiente interfaz para Ave:

```
interface IAve
{
    void Vuela();
    void Nada();
    void Come();
    void PonHuevo();
}
```

Ahora definimos las siguientes clases que implementan la interfaz.

class Pato : IAve { Vuela(), Come(), PoneHuevo(), Nada() }

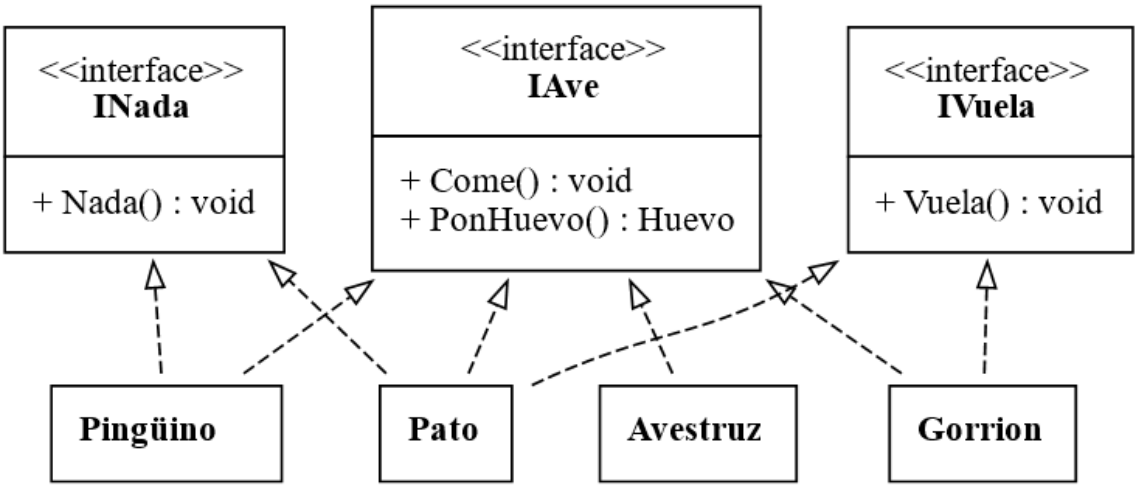
class Gorrion : IAve { Vuela(), Come(), PoneHuevo(), ~~Nada()~~ }

class Avestruz : IAve { ~~Vuela()~~, Come(), PoneHuevo(), ~~Nada()~~ }

class Pingüino : IAve { ~~Vuela()~~, Come(), PoneHuevo(), Nada() }

Si nos fijamos hay operaciones que no somos capaces de implementar, estamos obligados a definirlas por heredar del interfaz, pero seguramente generaremos algún tipo de excepción al llamarlas. En este caso no estaremos siguiendo el principio de ISP.

La mejor solución es segregar los interfaces y que cada Ave implemente aquellos que le corresponden.



7.5 Principio de Inversión de Dependencias (DIP)

Es uno de los más importantes y muy a tener en cuenta en nuestros diseños. El hacerlo, nos facilitará la labor de creación de test. Este principio establece que:

A. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Suponemos el siguiente ejemplo:

La clase de alto nivel [EstacionMeteorologica] depende de la clase de bajo nivel [Termómetro] (o Barómetro, Anemómetro, etc.). Toda la lógica de la solución se implementaría en la clase de alto nivel, y cualquier modificación en las clases de bajo nivel tendría repercusión no únicamente sobre la definición de la clase de alto nivel, sino sobre la propia lógica de la aplicación, llegando incluso a forzar cambios en la misma, cuando debería ser la clase de alto nivel la que debería forzar el cambio a las clases de bajo nivel sin comprometer la lógica de la aplicación; es decir, justamente lo contrario. Además, la clase de alto nivel sería difícilmente reusable debido a este acoplamiento. Sencillamente, y resumiendo, la clase [EstacionMeteorologica] no debe depender de la clase [Termómetro].

En otras palabras podemos decir que, el objetivo de este principio es el uso de interfaces para conseguir que una clase interactúe con otras clases sin que las conozca directamente y conseguir desacoplar así las clases lo máximo posible.

Existen diferentes patrones como **la inyección de dependencias** o service locator que nos permitirán invertir el control.

Supongamos la siguiente composición simple en la que un vehículo está compuesto por un motor.

```
public class Motor
{
   public void Acelera() {  }
   public int Revoluciones { get { return ..; } }
}
public class Vehiculo
{
   private Motor Motor { get; }

   public Vehiculo() { Motor = new Motor(); }
   public void Acelera() { Motor.Acelera(); }
   public int Revoluciones{get { return Motor.Revoluciones; }}
```

Tenemos claramente una clase de alto nivel Vehiculo que depende de otra de Bajo nivel Motor. Esto significa que la tiene que "conocer".

Para que cumpla el principio haremos que depende de una abstracción de Motor que denominaremos IMotor. De esta manera será más fácil que nuestro vehículo tenga distintos tipos de motor sin modificar la clase Vehiculo.

```
public interface IMotor
{
   void Acelera();
   int Revoluciones { get; }
}
```

Inyectaremos desde el constructor la dependencia con IMotor

```
public class Vehiculo
{
   private IMotor Motor { get; }
   public Vehiculo(IMotor motor) { Motor = motor; }
   public void Acelera() { Motor.Acelera(); }
   public int Revoluciones {
        get { return Motor.Revoluciones; }}
}
```

Ahora podremos definir Motores que implementen la abstracción:

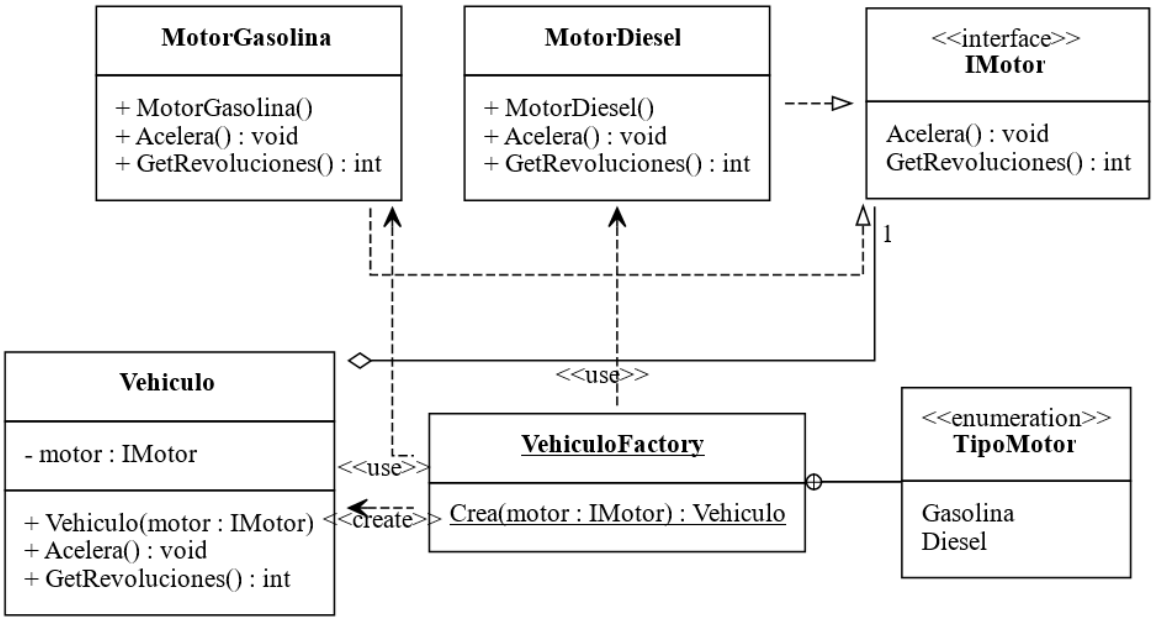
```
public class MotorGasolina : IMotor
{
   public void Acelera()
   {
      Admision(); Compresion(); Explosion(); Escape();
   }
   public int Revolucione { get { return …; } }
   //…
}
public class MotorDiesel : IMotor
{
   public void Acelera()
   {
      Admision(); Compresion(); Combustion(); Escape();
   }
   public int Revoluciones { get { return …; } }
   //…
}
```

Pero cómo realizar esta **inyección de dependencias** teniendo en cuenta la composición entre Vehículo y Motor. Usaremos el patrón **Simple Factory** el cual se encargará de relacionar las dos clases.

```
public static class VehiculoFactory
{
    public enum TipoMotor { Gasolina, Diesel }
    public static Vehiculo Crea(TipoMotor tipo)
    {
       Vehiculo v = null;
       switch (tipo)
       {
          case TipoMotor.Diesel:
              v = new Vehiculo(new MotorDiesel());
              break;
           case TipoMotor.Gasolina:
              v = new Vehiculo(new MotorGasolina());
              break;
           default:
              throw new ArgumentException("Motor no definido");
        }
        return v;
     }
}
```

Si nos fijamos a efectos prácticos tendremos la composición porque las referencias a motor desaparecerán junto con el vehículo.
Ahora ya podremos instanciar vehículos usando nuestra clase factoría de la siguiente manera:

```
Vehiculo vg = VehiculoFactory.Crea(VehiculoFactory.TipoMotor.Gasolina);
Vehiculo vd = VehiculoFactory.Crea(VehiculoFactory.TipoMotor.Diesel);
```


MotorGasolina
+ MotorGasolina()
+ Acelera() : void
+ GetRevoluciones() : int
MotorDiesel
+ MotorDiesel()
+ Acelera() : void
+ GetRevoluciones() : int
<<interface>>
IMotor
Acelera() : void
GetRevoluciones() : int
1
<<use>>
Vehiculo
- motor : IMotor
+ Vehiculo(motor : IMotor)
+ Acelera() : void
+ GetRevoluciones() : int
<<use>>
<<create>>
VehiculoFactory
Crea(motor : IMotor) : Vehiculo
<<enumeration>>
TipoMotor
Gasolina
Diesel