POO

MODULARIZANDO Y ORGENIZANDO NUESTRAS CLASES

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Empaquetando Clases

- Ya hemos visto que clases relacionadas se pueden agrupar bajo un espacio de nombres (namespace) al que podremos denominar también "Paquete de clases".

```
namespace IdPaquete {
     public class ClaseA { … }
     public class ClaseB { … }
}
```

- También podemos hacer subagrupaciones anidando definiciones de espacios de nombres.

```
namespace IdPaquete {
     public class ClaseA { … }
     public class ClaseB { … }

     namespace IdSubPaquete {
          public class ClaseC { … }
          public class ClaseD { … }
     }
}
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

- Una sintaxis equivalente a la anterior podría ser:

```
namespace IdPaquete {
     public class ClaseA { … }
     public class ClaseB { … }
}

namespace IdPaquete.IdSubPaquete {
     public class ClaseC { … }
     public class ClaseD { … }
}
```

- Además, podremos repetir el identificador o nombre de una clase en paquetes diferentes.

```
namespace IdPaquete1 {
     public class ClaseA { … }
}

namespace IdPaquete2 {
     public class ClaseA { … }
}
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Accediendo A Un Paquete De Clases

- Para acceder a las clases públicas de un paquete, desde otro paquete podremos usar el nombre completamente cualificado (CCN), esto es, indicando la ruta separada por puntos

```
namespace MiPaquete {
 public class MiClase {
     IdPaquete.ClaseA a = new IdPaquete.ClaseA();
     IdPaquete.SubPaquete.ClaseD d = new IdPaquete.SubPaquete.ClaseD();
…
```

- O podremos usar la cláusula using <NombreDelPaquete>; normalmente al principio del fuente y donde indicaremos aquellos espacios de nombres o paquetes de los que queramos utilizar sus clases públicas.

```
using IdPaquete;
using IdPaquete.SubPaquete;

namespace MiPaquete {
 public class MiClase {
     ClaseA a = new ClaseA(); // Definida en IdPaquete
     ClaseD d = new ClaseD(); // Definida en IdPaquete.SubPaquete
…
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Accediendo A Un Paquete De Clases

- También podemos crear alias que nos facilitas el acceso a espacios de nombres o a Tipos sobre todo cuando el id está repetido.

```
using ClaseA1 = IdPaquete1.ClaseA;
using ClaseA2 = IdPaquete2.ClaseA;

namespace MiPaquete
{
     public class MiClase
     {
          ClaseA1 a1 = new ClaseA1();
          Clasea2 a2 = new ClaseA2();
     }
}
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Separando En 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.

Archivos Generados

- 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.

Gestores De Paquetes

- Podré publicar mis 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.

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Concepto De Dependencia

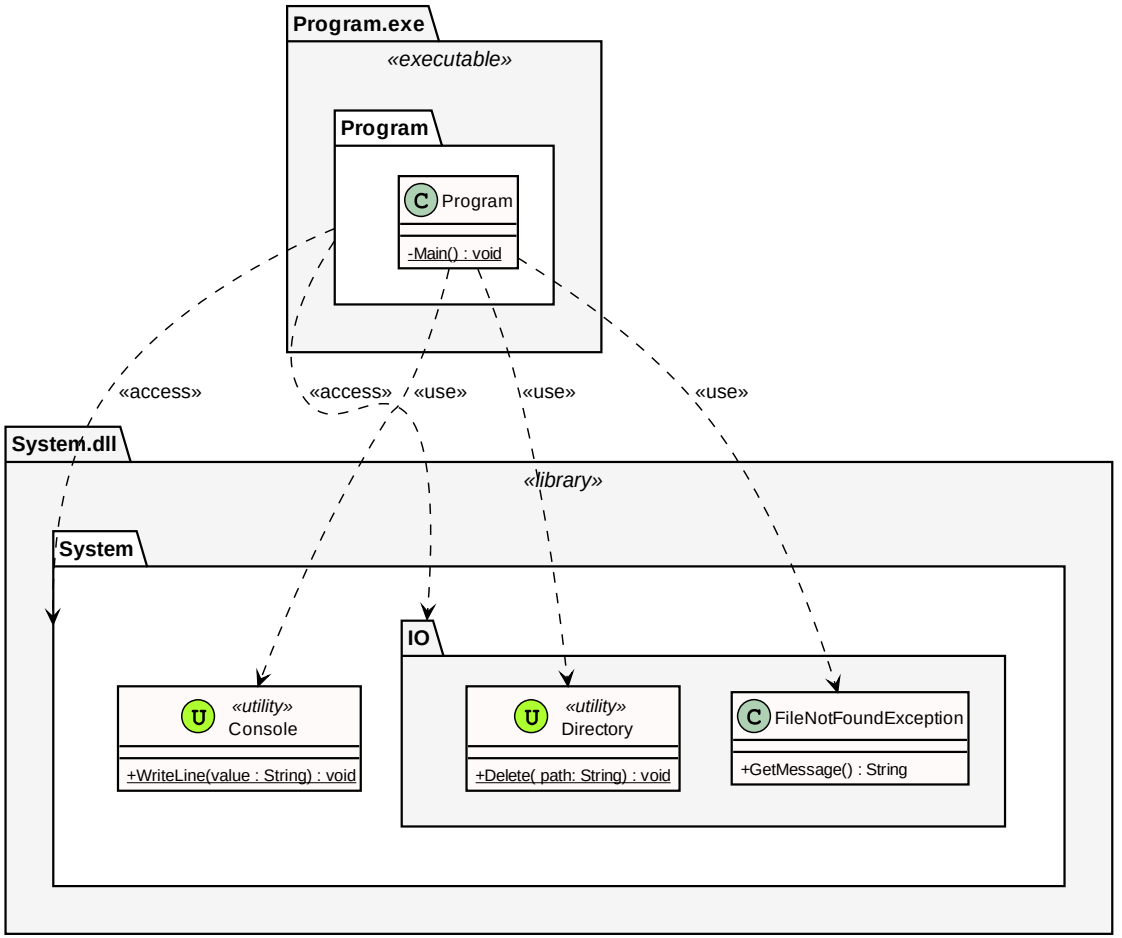
- Hasta ahora en un programa sencillo en C# como el del ejemplo hemos usados librerías de clases de las BCL.

```
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);
            }
        }
    }
}
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Concepto De Dependencia

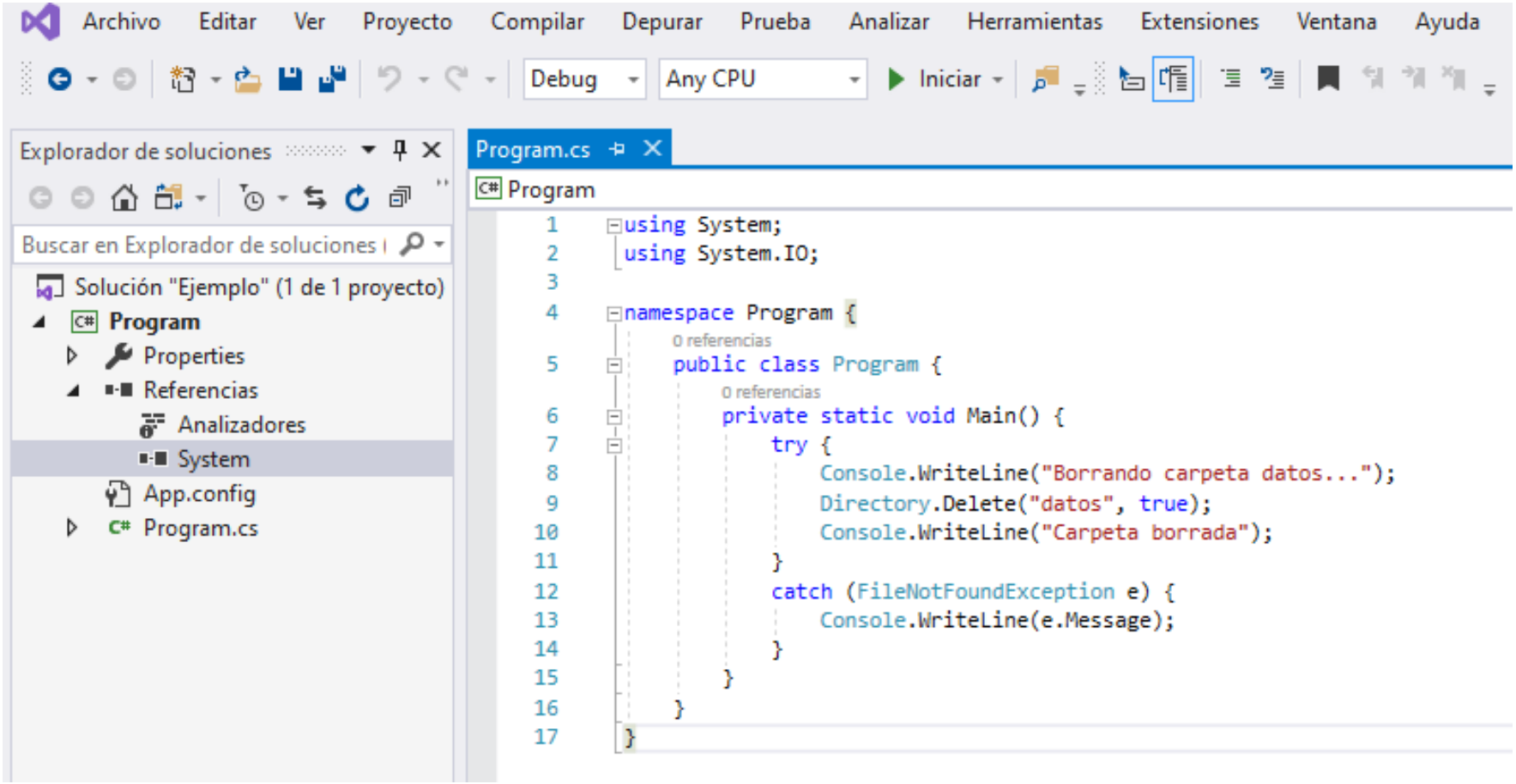
- Del ejemplo anterior se nos habrá generado una dependencia entre clases ya que nuestra clase `Program` usa otras clases como `Console`, `Directory` y `FileNotFoundException` y además se ha generado una dependencia.
- Además tendremos una dependencia entre librerías o ensamblados. Ya que las clases que usamos están definidas en los paquetes `System` y `System.IO` dentro de System.dll



ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Concepto De Dependencia

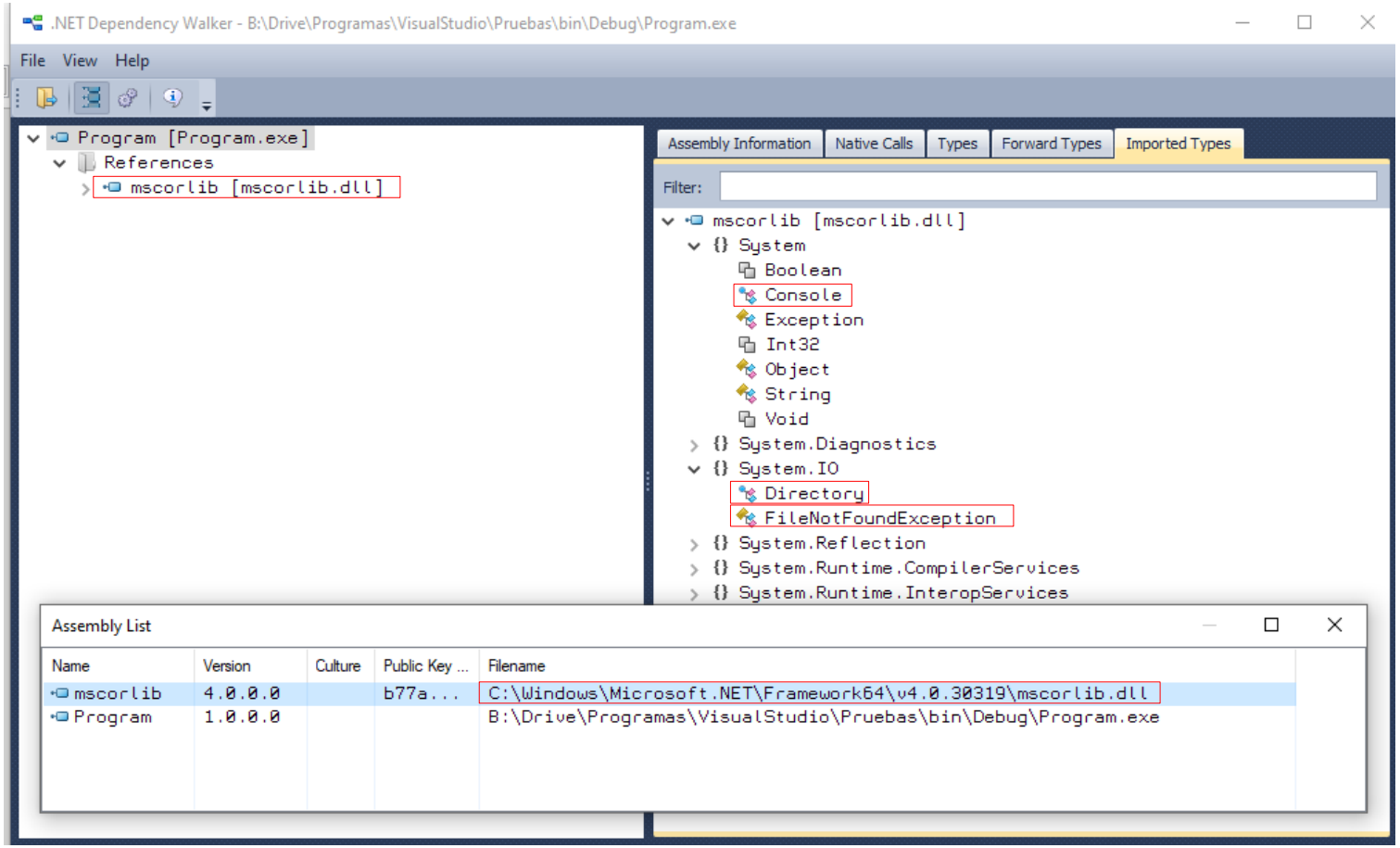
- Podemos ver y establecer las dependencias de nuestro programa a través del apartado referencias en nuestro proyecto.



ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Concepto De Dependencia

- Además, podremos consultar las dependencias de nuestros programas con herramientas como .NET Dependency Walker. Si examinamos Program.exe:



ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Usando Librerías De Terceros

- Vamos a verlo a través de un ejemplo. En nuestro caso queremos comprobar ortográficamente una palabra introducida por teclado.
- Crearemos un proyecto y iremos por ejemplo a Ver → Otras Ventanas → Consola del Administrador de Paquetes y usaremos el comando Find-Package para buscar algún paquete de terceros.

```
PM> Find-Package hunspell

Id                                Versions                          Description
--                                --------                          -----------
WeCantSpell.Hunspell              {3.0.1}                           A port of Hunspell
…
```

- Para instalarlo en nuestro proyecto ejecutaremos Install-Package <Id>

```
PM> Install-Package WeCantSpell.Hunspell
…
```

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

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Usando Librerías De Terceros

- 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.
- Tras compilar para Debug. Podremos comprobar que se han añadido la librería WeCantSpell.Hunspell justo a nuestro programa principal y además la versión específica de todas las librerías que tiene la misma como dependencias y que deberíamos incluir en el despliegue de nuestra aplicación. (Podemos comprobar estas dependencias con el Dependency Walker). Si faltara alguna de estas librerías nuestra aplicación no funcionaría.
- Si quisieramos quitar la librería de nuestro proyecto Uninstall-Package <Id>

```
PM> Uninstall-Package WeCantSpell.Hunspell
…
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Usando Librerías De Terceros

```
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");
    }
}
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Creando Nuestras Propias Librerías

- Con Visual Studio el proceso es bastante sencillo.

1. Agregaremos un nuevo proyecto a nuetra solución del tipo:
   Biblioteca de clases (.NET Framework)
   Le podemos llamar por ejemplo `MiLibreria`
2. 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 por defecto.

   Además, podremos definir otros subespacios de nombres para agrupar nuestras clases en paquetes más específicos. En algún momento algúno de estos subpaquetes podría ir a una nueva librería.
3. Vamos añadir una clase de utilidad para ampliar las funcionalidades desde la entrada por la consola. Para ello copia el código siguiente….

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

```
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");
                    }
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

```
                    else if (key.Key == ConsoleKey.Enter)
                    {
                        Console.Write("\n");
                        break;
                    }
                }
            } while (true);
            return text;
        }
        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;
        }
    }
}
```

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

Creando Nuestras Propias Librerías

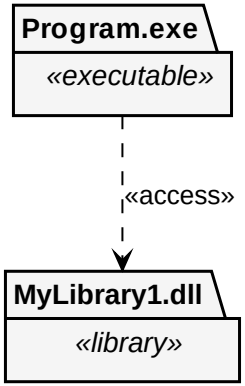
4. Ahora en nuestro proyecto donde está el programa principal vamos incluir la dependencia con el proyecto donde está la librería.
   Botón derecho sobre Referencias → Agregar referencia y podremos:
   - Indicar el proyecto con la librería que queremos usar (Recomendado).
   - Examinar en disco, en busca del ensamblado con la librería.
5. Ahora ya podremos usar la librería en nuestro programa:

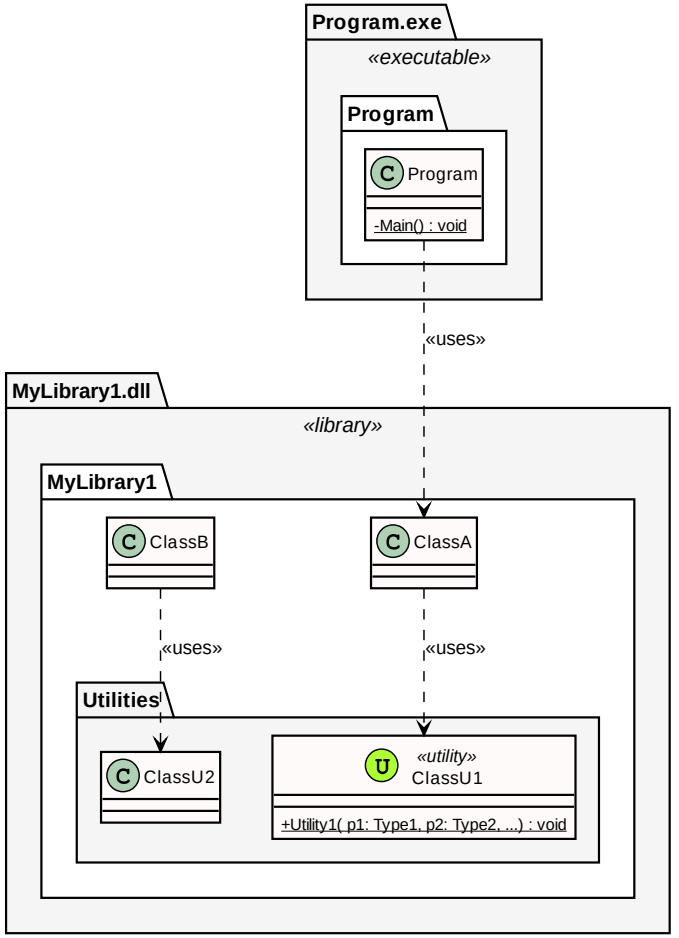
```
using System;
using MiLibreria;

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

ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

¿Cómo Gestionar Las Dependencias Entre Librerías?

- Vamos a verlo a través de un ejemplo.
- 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…

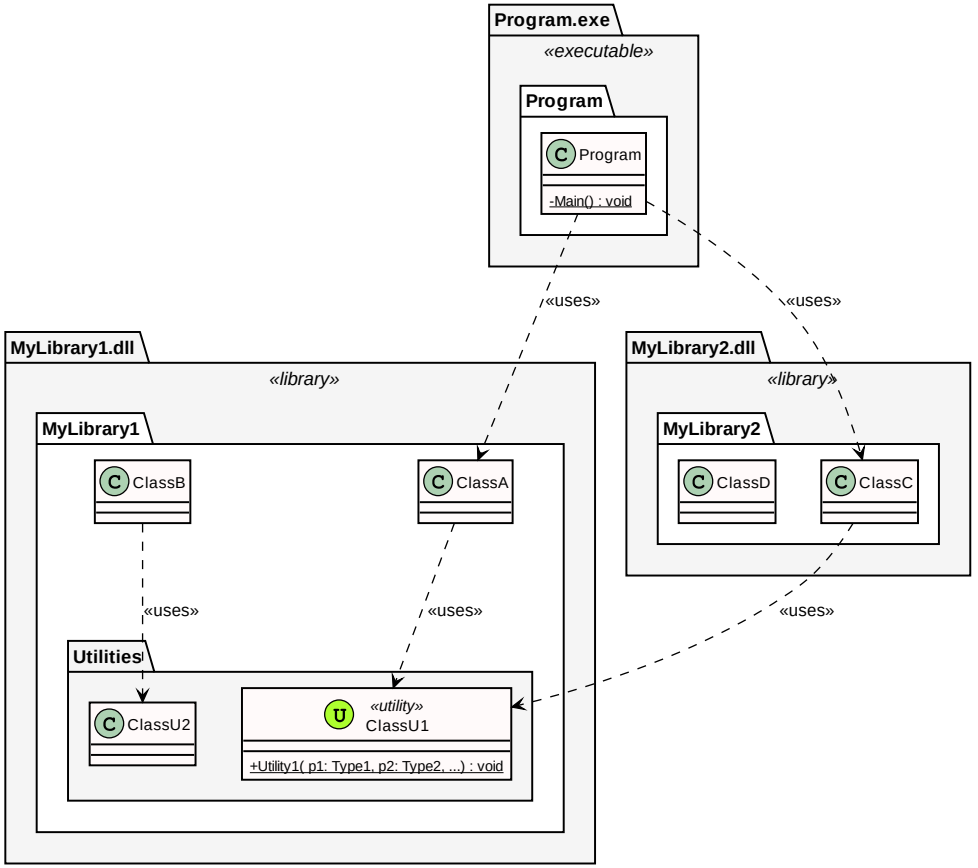




ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

¿Cómo Gestionar Las Dependencias Entre Librerías?

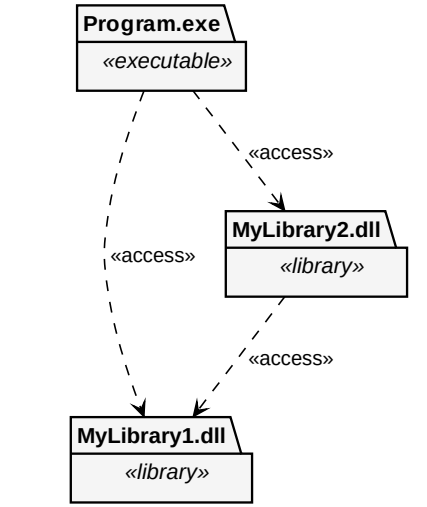
- Ahora desde la clase ClassC en una segunda librería MyLibrary2.dll también queremos utilizar el método de utilidad en `MyLibrary1.Utilities.ClassU1`
- Se producirá una situación de doble ámbito de uso de la clase de utilidad. Pues es usado dentro del la propia MyLibrary1.dll y ahora fuera por parte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

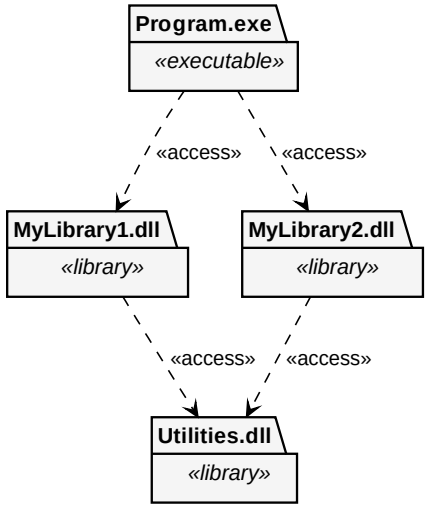


ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

¿Cómo Gestionar Las Dependencias Entre Librerías?

- Este doble ámbito, generará una dependencia de MyLibrary1.dll por parte de MyLibrary2.dll cuando realmente solo hay una clase que queremos reutilizar y estaba ahí en un paquete porque hasta ahora solo se utilizaba ahí.
- Ahora siempre que queramos usar MyLibrary2.dll vamos a necesitar que esté instalada MyLibrary1.dll cuando hay un montón de clases que no vamos a utilizar ni nos interesan de la misma.
- La solución será sacar la clase que produce el doble ámbito de uso a una tercera librería, por ejemplo Utilities.dll lo que estaba produciendo la dependencia no adecuada. Bajando así el acoplamiento elevado que se estaba produciendo por el doble ámbito.

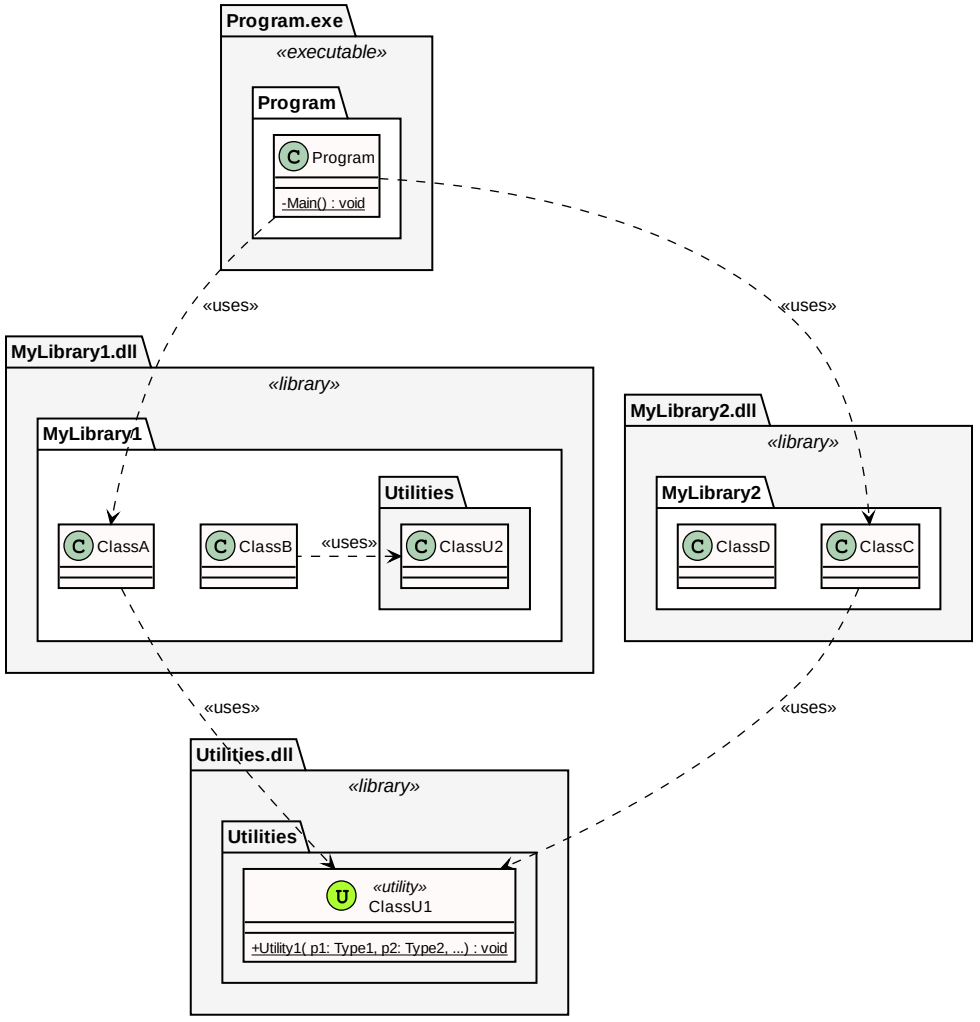




ORGANIZANDO EN PAQUETES O ESPACIOS DE NOMBRES

¿Cómo Gestionar Las Dependencias Entre Librerías?

- 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.



POO

DIFERENTES POLIMORFISMOS DIFERENTES FORMAS

1 - POLIMORFISMO DATOS O INCLUSIÓN

- Se basa en el Upcasting o Principio de Sustitución de Liskov que ya vimos al estudiar el concepto de herencia y el downcasting.
- Pero además de estas tendremos otras formas de polimorfismo...

2 - POLIMORFISMO FUNCIONAL O SOBRECARGA

- Al igual que el de datos, ya lo hemos visto y usado con anterioridad.
- Es la capacidad de definir operaciones o métodos con el mismo identificador o nombre. Siempre y cuando, la signatura cambie.
- En C# dos métodos tienen diferente signatura si:
  - Tienen diferente tipo de retorno.
  - Tienen diferente número de parámetros.
  - Teniendo el mismo número de parámetros y algún tipo es diferente.
  - Teniendo el mismo número de parámetros y el mismo tipo alguno tiene el modificador ref o out.

2 - POLIMORFISMO FUNCIONAL O SOBRECARGA

Ejemplos

```
Class SobrecargaValida
{
     public void MetodoA(int x) {
          ;
     }

     public void MetodoA(ref int x) {
          ;
     }
}
```

```
Class SobrecargaInvalida
{
     public void MetodoA(out int x) {
          ;
     }

     public void MetodoA(ref int x) {
          ;
     }
}
```

2 - POLIMORFISMO FUNCIONAL O SOBRECARGA

¿Para Qué Se Usa?

- Por ejemplo, lo venimos usando para tener varios constructores de un objeto.
- Para evitar el uso de parámetros opcionales o por defecto en los métodos.
  - C# no los permitía en sus primeras versiones.
  - Java no los permite y Sacala, php, Javascript, Python… sí.
  - Su uso no es recomendable porque:
    - Pude dar lugar a baja cohesión (Un método hace muchas cosas).
    - Ralentizan la ejecución.
    - Lleva a confusión a los usuarios de una clase.
  - No deberíamos usarlos en **métodos públicos**.

2 - POLIMORFISMO FUNCIONAL O SOBRECARGA

Parámetros Opcionales O Por Defecto - I

- Recordemos nuestra clase PuntoConsola y su método desplaza.

```
Class PuntoConsola
{
    …
    public void Desplaza(ushort numPosiciones, double angulo = 0d) {
        double anguloRad = angulo * Math.PI / 180d;
        fila += (ushort)(numPosiciones * Math.Sin(anguloRad));
        columna += (ushort)(numPosiciones * Math.Cos(anguloRad));
    }
}

PuntoConsola p = new PuntoConsola(2,4);

// Como el parámetro formal  angulo es opcional, podremos llamar al
// método Desplaza sin especificarlo.
// Es cuyo caso tomará su valor por defecto
// 0 grados desplzando el punto 4 columnas a la derecha.
p.Desplaza(4);
```

¿Cómo Deberíamos Realizar El Código Anterior Usando El Polimorfismo Funcional O Sobrecarga?

2 - POLIMORFISMO FUNCIONAL O SOBRECARGA

Parámetros Opcionales O Por Defecto – II

```
Class PuntoConsola
{
    private void desplaza(ushort numPosiciones, double angulo) {
        double anguloRad = angulo * Math.PI / 180d;
        fila += (ushort)(numPosiciones * Math.Sin(anguloRad));
        columna += (ushort)(numPosiciones * Math.Cos(anguloRad));
    }

    public void Desplaza(ushort numPosiciones) {
        desplaza(numPosiciones, 0d);
    }

    public void Desplaza(ushort numPosiciones, double angulo) {
        desplaza(numPosiciones, angulo);
    }
}
PuntoConsola p = new PuntoConsola(2,4);
p.Desplaza(4);
```

3 - POLIMORFISMO PARAMÉTRICO O TIPOS GENÉRICOS

- Lo trataremos en temas posteriores

POO
ENCAPSULACIÓN

ENCAPSULACIÓN

Objetivos

- Evitar que un cliente de mis clases puedan dejar objetos instanciados de las mismas en un estado inadecuado.
- Ocultar detalles de la implementación de una clase.
- Realizar cambios o actualizaciones en la clases sin preocuparnos cómo están siendo usadas.
- Ya hemos usado la encapsulación, marcando los atributos como privados y a través del uso de accesores y mutadores.

ENCAPSULACIÓN

Recordemos los modificadores de acceso que hay para clases, tipos, atributos y métodos.

- private: Accesible solo desde la clase. Es lo que debemos poner por defecto.
- public: Accesible por todos.
- protected: Accesible solo desde la clase o las subclases.
- internal: Accesible solo desde clases del ensamblado o paquete actual.
- protected internal: Accesible solo desde clases del paquete actual y además sean subclases de la clase donde se ha definido.

ENCAPSULACIÓN

Propiedades En C# - I

- Hasta ahora hemos utilizado la sintaxis de otros lenguajes como Java, C++ , php, etc… para definir los accesores y mutadores.

```
<Tipo> get<IdAtributo>() {
     return <idAtributo>
}
void set<IdAtributo>(<Tipo> <idAtributo>) {
     this.<idAtributo>  = <idAtributo>
}
```

- Las propiedades son un "azucar sintáctico" incluido por el lenguaje C# para usarlos y de esta manera implementar la encapsulación.

```
<Tipo> <IdAtributo> {
   set {
      <idAtributo> = value;
   }
   get {
      return <idAtributo>;
   }
}
```

ENCAPSULACIÓN

Propiedades En C# - II

- Por ejemplo, para nuestro atributo fila en la clase PuntoConsola hacíamos….

```
public void SetFila(ushort fila) {
     if (fila > 24)
         throw new ArgumentException("fila > columna 24");
     this.fila = fila;
}
private ushort getFila() {
     return fila;
}
```

- Si los definimos como una propiedad en C# haremos…

```
public ushort Fila {
     set {
         if (value > 24)
             throw new ArgumentException("fila > columna 24");
         fila = value;
     }
     private get {
         return fila;
     }
}
```

ENCAPSULACIÓN

Propiedades En C# - III

- Podremos definir solo uno de los dos, set o get y estarán afectadas por los modificadores de accesibilidad como el resto de métodos.

¿ Cómo Usaremos La Propiedad Cuando Instanciemos Un Objeto De La Clase PuntoConsola ?

- Para nosotros sintácticamente, es como si estuvieramos accediendo directamente al atributo.

```
PuntoConsola punto = new PuntoConsola(2, 4);

Console.WriteLine(punto.Fila); // Solo se podrá acceder a la
                               // propiedad Fila dentro de la clase.
                               // Por estar marcado el set como private.

punto.Fila = 12; // Cambiaremos el valor de la fila a 12.
                 // Internamente se llama al set

punto.Fila = 25; // Generará un ArgumentException
```

ENCAPSULACIÓN

Propiedades Autoimplementadas A Partir De C# 3.0 - IV

- A partir de C# 3.0 , 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.
- https://msdn.microsoft.com/es-es/library/bb384054.aspx

REDEFINICIÓN DE OPERADORES

REDEFINICIÓN DE OPERADORES

- A principio de este curso, vimos cómo se evaluaban expresiones y los operadores unarios, binarios, de cast, etc… usados con tipos simples.
- Para dos enteros la suma binaria + los sumaba …
  2 + 3 = 5
- Pero para dos cadenas esta suma binaria + significaba concatenación…
  "Hola " + "Caracola" = "Hola Caracola"

Entonces... ¿Podemos Cambiar El Sentido De La Suma Según El Tipo?

- La respuesta es Sí.
- En los lenguajes orientados a objetos, podremos darle el significado que queramos al operador + y al resto de operadores, cuando lo apliquemos a dos objetos de una clase definida por nosotros.

- Ojo !!! Siempre y cuando la operación tenga sentido.

REDEFINICIÓN DE OPERADORES

¿Cómo Se Redefinen Operadores En C#?

- Usaremos la siguiente sintaxis.

```
public static <tipoDevuelto> operator <simboloOperqcion> (<operandos>)
{
    <cuerpo de método de clase>
}
```

- 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.

REDEFINICIÓN DE OPERADORES

¿Cómo Se Redefinen Operadores En C#?

- Supongamos la siguiente clase que representa 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;
    }
}
```

- Vamos a implementar el operador binario + que sumará dos números complejos.

REDEFINICIÓN DE OPERADORES

Redifinición De La Suma Binaria Para La Clase Complejo

```
class Complejo {
     …
     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);
     }
}
```

- Ya podremos sumar complejos de la siguiente forma ...

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

REDEFINICIÓN DE OPERADORES

Caso Específico De Los Operadores De Cast

- Operador de cast explícito.

```
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.
```

- Operador de cast implícito.

```
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
```

REDEFINICIÓN DE OPERADORES

Caso Específico Del Operador Unario De Pre Y Post Incremeto/Decremento

- Cuando se usen de forma prefija se evaluará el nuevo objeto creado, y cuando se usen de forma postifja el compilador lo que hará será evaluar la referencia al objeto original que se les pasó como parámetro en lugar del creado en el return.
- En ambos casos tras la evaluación c pasará a referenciar al objeto creado en el return.

```
public static Complejo operator ++ (Complejo c) {
     return new Complejo(c.ParteReal + 1, c.ParteImaginaria);
}
…
Complejo c = new Complejo(1d, 1d);
// cAux referenciará a c y después c será el nuevo obajeto incrementado.
Complejo cAux = c++;

// c será el nuevo obajeto incrementado y cAux referenciará al nuevo c.
Complejo cAux = ++c;
```

INDIZADORES

INDIZADORES

- Es un concepto muy parecido al de las propiedades.
- Me permiten mapear o indexar datos de un determinado tipo dentro de una clase utilizando el doble corchete [] como en los arrays.
- Este operador lo aplicaremos a los objetos instanciados de dicha clase.
- Para indexar no necesariamente utilizaremos un entero, también podremos utilizar otros tipos como cadenas, tipos enumerados, etc…

Sintaxis

```
<tipoDelIndizador> this[<parámetrosQueSirvenDeÍndice>]
{
    set {
        // Código para cambiar un "dato" del objeto según los índices.
    }
    get {
        // Código para acceder a un "dato" del objeto según los índices.
    }
}
```

INDIZADORES

Ejemplo

- Supongamos la siguiente clase que contendrá valores de habilidades de Humano de tal manera que cada una estará descrita por un tipo enumerado.

```
class HabilidadesHunamo
{
    public enum Habilidad { Inteligencia, Fuerza, Destreza, Energia }

    private ushort[] valores;

    public HabilidadesHunamo()
    {
        valores = new ushort[Enum.GetNames(typeof(Habilidad)).Length];
    }

    public override string ToString()
    {
        string texto = "";
        string[] habilidades = Enum.GetNames(typeof(Habilidad));
        for (int i = 0; i < habilidades.Length; i++)
            texto += $"{habilidades[i]} = {valores[i]}\n";
        return texto;
    }
}
```

INDIZADORES

- Vamos a definirle un indizador que me va a permitir cambiar el valor de la habilidad a través del tipo enumerado con la habilidad.

```
class HabilidadesHunamo
{
     …
     public ushort this[Habilidad habilidad]
     {
         get { return valores[(int)habilidad];  }
         set { valores[(int)habilidad] = value; }
     }
}
```

- Aunque este esquema no es muy “ortodoxo”. ¿Qué piensas que hemos ganado con la definición esta clase respecto a una colección normal?

```
static void Main()
{
    HabilidadesHunamo habilidadesHunamo = new HabilidadesHunamo();
    habilidadesHunamo[HabilidadesHunamo.Habilidad.Destreza] += 10;
    habilidadesHunamo[HabilidadesHunamo.Habilidad.Energia] += 5;
    Console.WriteLine(habilidadesHunamo);
}
```

POO

INTERFACES

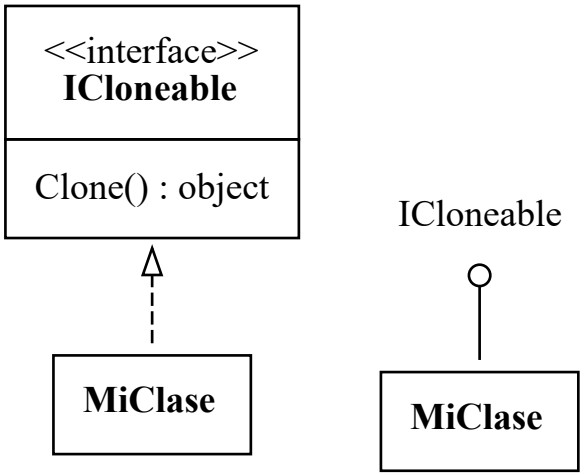
INTERFACES DEFINICIÓN

- Básicamente un Interfaz es la definición de un conjunto de interfaces de métodos, accesores o mutadores (propiedades), indizadores, etc.
- Es muy parecido a definir una clase abstracta pura, pero sin ningún tipo de atributo, constructor, ni modificador de acceso (public, private, etc…).
- Pueden implementarse en muchos lenguajes OO 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 lo que hayamos definido en el mismo.
- Podemos resumir diciendo que los interfaces definen un comportamiento.

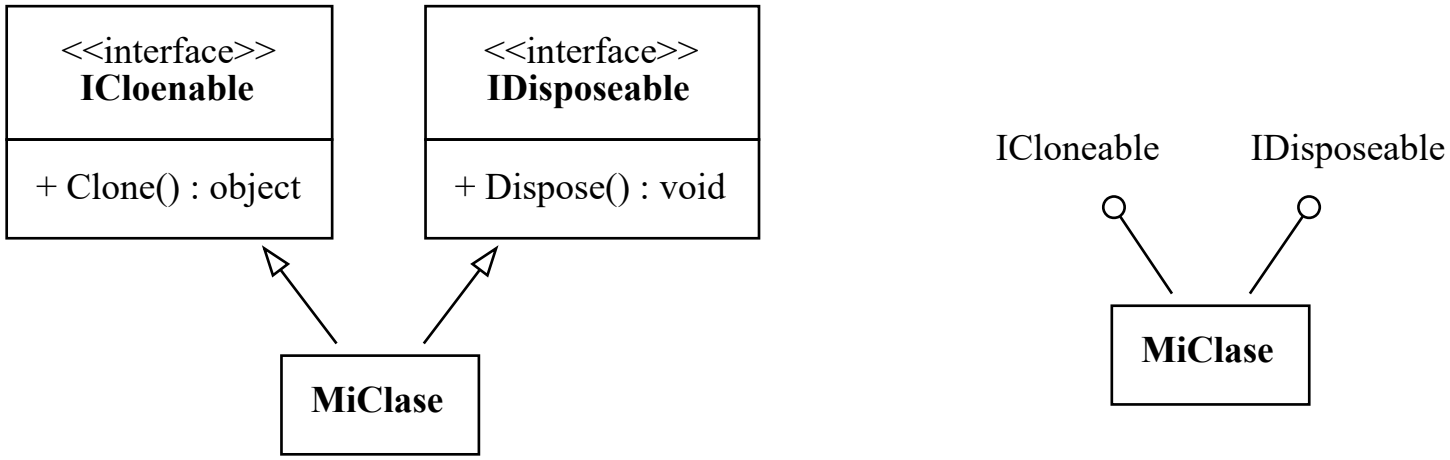
INTERFACES EN LOS DIAGRAMAS DE CLASES UML

- Para expresar que la clase MiClase implementa el interfaz ICloneable.
  Podremos expresarlo de las forma siguiente →

  MiClase ahora está obligada a implementar el método público Clone con idéntica signatura.



Podremos Hacer Que Una Clase Implemente O "herede" De Más De Un Interfaz

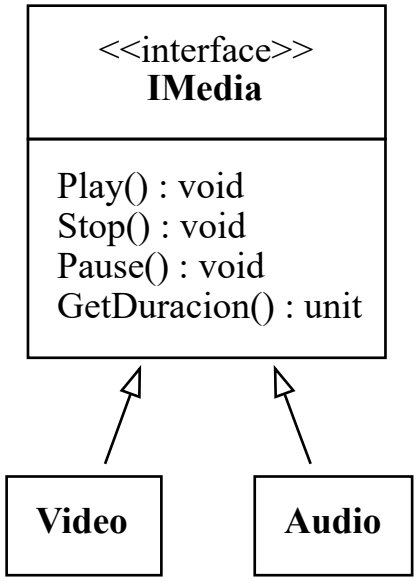


INTERFACES SINTAXIS Y DEFINICIÓN EN C# - I

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

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

Ejemplo Definición Del Interface IMedia




```
interface IMedia
{
    void Play();
    void Stop();
    void Pause();
    uint Duracion { get; }
}
```

INTERFACES SINTAXIS Y DEFINICIÓN EN C# - II

Ejemplo Implementación Del Interface IMedia

- Para aplicar un interfaz a una clase. Haremos que esta herede del interfaz con la sintaxis de herencia que hemos usado hasta ahora.

```
class Video : IMedia {
    …
    public uint Duracion {
        get { return 0; }
    }

    public void Pause() {
        Console.WriteLine("Pausando el vídeo.");
    }

    public void Play() {
        Console.WriteLine("Reproduciendo el vídeo.");
    }

    public void Stop() {
        Console.WriteLine("Parando el vídeo.");
    }
}
```

ALGUNOS INTERFACES PREDEFINIDOS EN LAS BCL - I

IDisposable (MSDN)

- Me indicará que el objeto debe implementar el método void Dispose() que se encargará de liberar los recursos usados por el objeto.
  Ojo no confundir con el destructor ~<Tipo>()

ICloneable (MSDN)

- Me indicará que puedo crear copias del objeto, puesto que me obliga a implementar un “constructor copia” con el interfaz Object Clone()
- Me permitirá hacer copias en profundidad.

IComparable (MSDN)

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

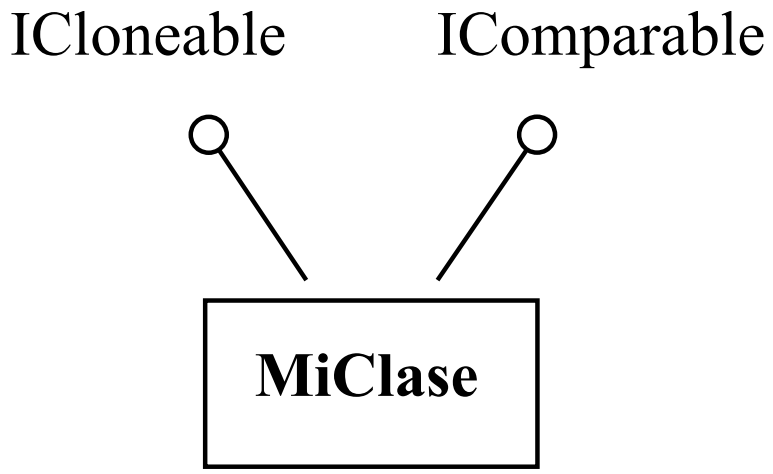
IEnumerable

- Lo veremos posteriormente al usar o definir colecciones.

ALGUNOS INTERFACES PREDEFINIDOS EN LAS BCL – II

Herencia Múltiple De Interfaces

```
class MiClase : ICloneable, IComparable
{
    public int MiPropiedad {
        get; set;
    }

    private MiClase(MiClase c) {
        MiPropiedad = c.MiPropiedad;
    }

    protected virtual object clone() {
        return new MiClase(this);
    }

    public object Clone() {
        return clone();
    }

    public int CompareTo(object obj) {
        int comparacion;
        if (obj == null) {
            comparacion = 1; // También podemos generar una excepción.
        }
```



```
            …
            else {
                MiClase mc = obj as MiClase;

                if (mc == null) throw new ArgumentException(
                                        "El objeto a comparar no es de MiClase");

                if (MiPropiedad == mc.MiPropiedad)
                    comparacion = 0;
                else
                    comparacion = (MiPropiedad > mc.MiPropiedad) ? 1 : -1;
            }
            return comparacion;
        }
}
```

Resumen

Podríamos utilizar interfaces nuestros para hacer lo mismo, pero perderíamos interoperabilidad con el resto de clases de las BCL.
Podemos decir de alguna forma, que es como definir estándares para los interfaces de ciertas operaciones.

INSTRUCCIÓN USING - I

- Se utiliza para instanciar objetos que contiene recursos no gestionados, esto es, que no son liberados por el recolector de basura.
- Estos objetos, deben implementar el interfaz IDisposable y por tanto el método de liberación Dispose.
- 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.

Sintaxis

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

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

INSTRUCCIÓN USING – II

Nos asegura la liberación del recurso por la siguiente equivalencia….

```
using (TipoIDisposabe r = … )
{
   // Bloque...
}
```

```
TipoIDisposabe r;
try
{
   r = … ;// Bloque...
}
finally
{
   if (r != null)
       ((IDisposable)r).Dispose();

}
```

En C# 8.0 (2019) la funcionalidad de using se ha ampliado. Podremos aprovechar un bloque ya definido para decidir cuando va a etar displonible un recurso.

```
if (…)
{
   using TipoIDisposabe r = …
   // Bloque...
}
```

```
if (…)
{
   using (TipoIDisposabe r = … )
   {
       // Bloque...
   }
}
```

INSTRUCCIÓN USING – III

Ejemplo

- 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, using garantiza que el archivo subyacente se cierre correctamente después de las operaciones de lectura o escritura.

POO
GENÉRICOS
O
CLASES
PARAMETRIZADAS

CONSIDERACIONES PREVIAS

- Nos evitarán problemas en tiempo de ejecución derivados de un mal uso del tipo Object.
- Aclara y reducen cantidad de código repetido.
- Expresaremos el tipo o los tipos genéricos a través de una o más letras mayúsculas usadas a los 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 esta letras se les denomina: Parametros Tipo
- Podremos usarlos en clases, métodos, interfaces, etc…

Polimorfismo Paramétrico

- Permite definir el tipos dentro de una clase de forma parametrizada al instanciar un objeto de la misma. De tal manera que, para objetos diferentes el tipo con el que se instancia podrá cambiar.

SINTAXIS DE CLASES PARAMETRIZADAS EN C#

Definición

- Definiremos los tipos genéricos justo después del identificador de la clase entre < >.

```
public class A<T> {
    private T dato;

    public A(T dato) {
        this.dato = dato;
    }
}
```

Uso

- Cuando yo instáncie un objeto de la clase genérica A …

```
A<int> objA = new A<int>(4);
```

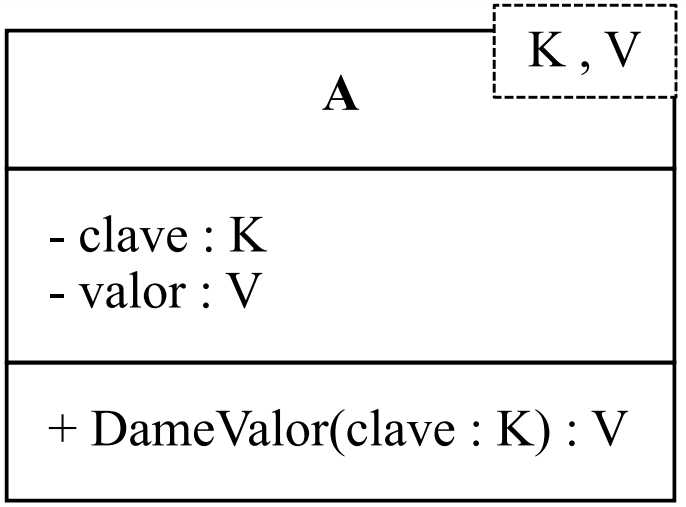
- En tiempo de ejecución C# construirá una objeto de la clase A como si su definición fuese la siguiente …

SINTAXIS DE CLASES PARAMETRIZADAS EN C#

Ejemplo De Definición Con Múltiples Parámetros

- Si vamos a usar más de un tipo para parametrizar. Los separaremos por comas.

```
public class A<K, V>
{
    private K clave;
    private V valor;

    public V DameValor(K clave)
    {
        ;
    }
}
```



CLASES PARAMETRIZADAS EN C#

¿Cómo Inicializar Un Dato Genérico A Su Valor Por Defecto?

```
class A<T>
{
    T dato = null; // Es correcto ¿?
}
```

- Usaremos la expresión default(T)

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

Expansión para int

```
void F<int>()
{
   int obj = 0;
}
```

Expansión para string

```
void F<string>()
{
   int obj = null;
}
```

Expansión para bool

```
void F<bool>()
{
   int obj = false;
}
```

CLASES PARAMETRIZADAS EN C#

Consideraciones Especiales - I

- 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 tendríamos 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.

CLASES PARAMETRIZADAS EN C#

Consideraciones Especiales - II

- Los parámetros tipo formarán parte de la signatura.

  Para C# en este ejemplo tendremos métodos diferentes.

```
class A
{
    public void Funcion(int p1, string p2) { ; }
    public void Funcion<T>(int p1, string p2) { T t = default(T);}
}
```

- 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.

CLASES PARAMETRIZADAS EN C#

Utilización De Genéricos En Lugar Del Tipo Object - I

- Supongamos que tenemos la clase Fecha implementa el interfaz IComparable, esto significa que:
  - Podremos pasar cualquier objeto, pues todos heredan de Object.
  - Deberemos hacer un downcast a fecha.
  - Además, tendremos que añadir un código de control de errores.

```
class Hora : Icomparable {
    private int hora, minuto;
    public int CompareTo(object objHora) {
        Hora hora = objHora as Hora;
        if (hora == null) throw new ArgumentException(
                                "El objeto a comparar no es una hora.");
        int comparacion = 0;
        if (hora.hora > this.hora) comparacion =  -1;
        else if (hora.hora < this.hora) comparacion =  1;
        else if (hora.minuto < minuto) comparacion =  -1;
        else if (hora.minuto > minuto) comparacion =  1;
        return comparacion;
    }
}
```

CLASES PARAMETRIZADAS EN C#

Utilización De Genéricos En Lugar Del Tipo Object - II

- Sin embargo si nosotros utilizamos la definición genérica de IComparable<T>
  - En tiempo de compilación nos avisará de que no estamos pasado el tipo correcto.
  - No necesitaremos hacer el repetitivo código de control anterior.

```
class Hora : IComparable<Hora>
{
    private int hora, minuto;
    …
    public int CompareTo(Hora hora)
    {
        int comparacion = 0;
        if (hora.hora > this.hora) comparacion =  -1;
        else if (hora.hora < this.hora) comparacion =  1;
        else if (hora.minuto < minuto) comparacion =  -1;
        else if (hora.minuto > minuto) comparacion =  1;
        return comparacion;
    }
}
```

CLASES PARAMETRIZADAS EN C#

Definición De Restricciones - I

- Podremos definir restricciones de tipo asociadas a su definición para los parámetros tipo.
- Se especifican con la cláusula …

  <T> where T : *restricción*

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

CLASES PARAMETRIZADAS EN C#

Definición De Restricciones - II

- Tendremos diferentes tipos de restricciones:
  - 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

POO
PRINCIPIOS
S.O.L.I.D.

PRINCIPIOS S.O.L.I.D.

Conceptos Básicos

- 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 fácil de mantener y escalable 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.

PRINCIPIOS S.O.L.I.D.

S – Principio De Responsabilidad Simple (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".

PRINCIPIOS S.O.L.I.D.

S – Principio De Responsabilidad Simple (SRP)

- Veamos un ejemplo sutil pero interesante de la aplicación de este principio.
- Supongamos que el estado de un objeto persona queremos formatearlo a cadena de diferentes formas para representarlo en diferentes sitios.

```
class Persona
{
    public string Nombre { get; private set;}
    public short Edad { get; private set; }
    …
    public string FomateaALinea() {
        return $"Nombre: {Nombre}, Edad: {Edad}";
    }
    public string FomateaATabla() {
        return $"{"Nombre",-8}{"Edad",-8}\n{Nombre,-8}{Edad,-8}";
    }
}
```

- Aparentemente no debería haber problema porque son operaciones sobre objetos de la clase, que acceden a sus propiedades.
- Pero, la clase ya es muy “pesada”, y por requerimientos, vamos a tener que añadir muchos métodos de Formato.

PRINCIPIOS S.O.L.I.D.

S – Principio De Responsabilidad Simple (SRP)

- Una posibilidad sería llevanos la responsabilidad del formateo a otras clases haciendo más “liviana” de la siguiente manera:

```
class FomateaPersonaATabla {
    private Persona Persona{ get; set; }
    public FomateaPersonaATabla(Persona persona) { … }

    public override string ToString() {
        return $"{"Nombre",-8}{"Edad",-8}\n" +
               $"{Persona.Nombre,-8}{Persona.Edad,-8}";
    }
}

class FomateaPersonaALinea {
    private Persona Persona { get; set; }
    public FomateaPersonaALinea(Persona persona) { … }

    public override string ToString() {
        return $"Nombre: {Persona.Nombre}, Edad: {Persona.Edad}";
    }
}
```

PRINCIPIOS S.O.L.I.D.

O – 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.

PRINCIPIOS S.O.L.I.D.

O – Principio De Abierto/Cerrado (OCP)

- 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; }
    }
}
```

PRINCIPIOS S.O.L.I.D.

O – Principio De Abierto/Cerrado (OCP)

- 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

  Definiremos la abstracción Ordenador. (Podría ser también un Interface)

```
public abstract class Ordenador {
    public abstract double Precio { get; }
}
```

  Definiremos decorador componente 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; }
}
```

PRINCIPIOS S.O.L.I.D.

O – Principio De Abierto/Cerrado (OCP)

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.

PRINCIPIOS S.O.L.I.D.

O – Principio De Abierto/Cerrado (OCP)

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}");
}
```

PRINCIPIOS S.O.L.I.D.

L – Principio De Sustitución De Liskov (LSP)

- Ya lo hemos usado y básicamente se trata de la importancia de definir bien sus subclases para que también puedan ser tratadas como la propia abstracción.
- Establece que las subclases deberían poder extender a su superclase sin cambiar el comportamiento de la misma.
- 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 ejemplo ilustrativo que nos generará problemas con el LSP:

- En la vida real tenemos “claro” que un cuadrado es un rectángulo con los dos lados iguales. Vamos a ver que pasa si intentamos implementar este modelo.

PRINCIPIOS S.O.L.I.D.

L – Principio De Sustitución De Liskov (LSP)

```
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 cuando cambiemos ancho y alto siga siendo 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; }
    }
}
```

PRINCIPIOS S.O.L.I.D.

L – Principio De Sustitución De Liskov (LSP)

- Ahora si pasamos el siguiente test...

```
private static void TestArea(Rectangulo r)
{
    r.Ancho = 5;
    r.Alto = 4;
    Debug.Assert(r.Area == 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.

```
TestArea(new Rectangulo());
TestArea(new Cuadrado());
```

- Una posible solución es hacer nuestro objeto inmutable, este es, no dar la posibilidad de que el Ancho y el Alto cambien después de su creación.

PRINCIPIOS S.O.L.I.D.

I – 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();
}
```

PRINCIPIOS S.O.L.I.D.

I – Principio De Segregación De Interfaces (ISP)

- Ahora definimos las siguientes clases que implementan la interfaz.

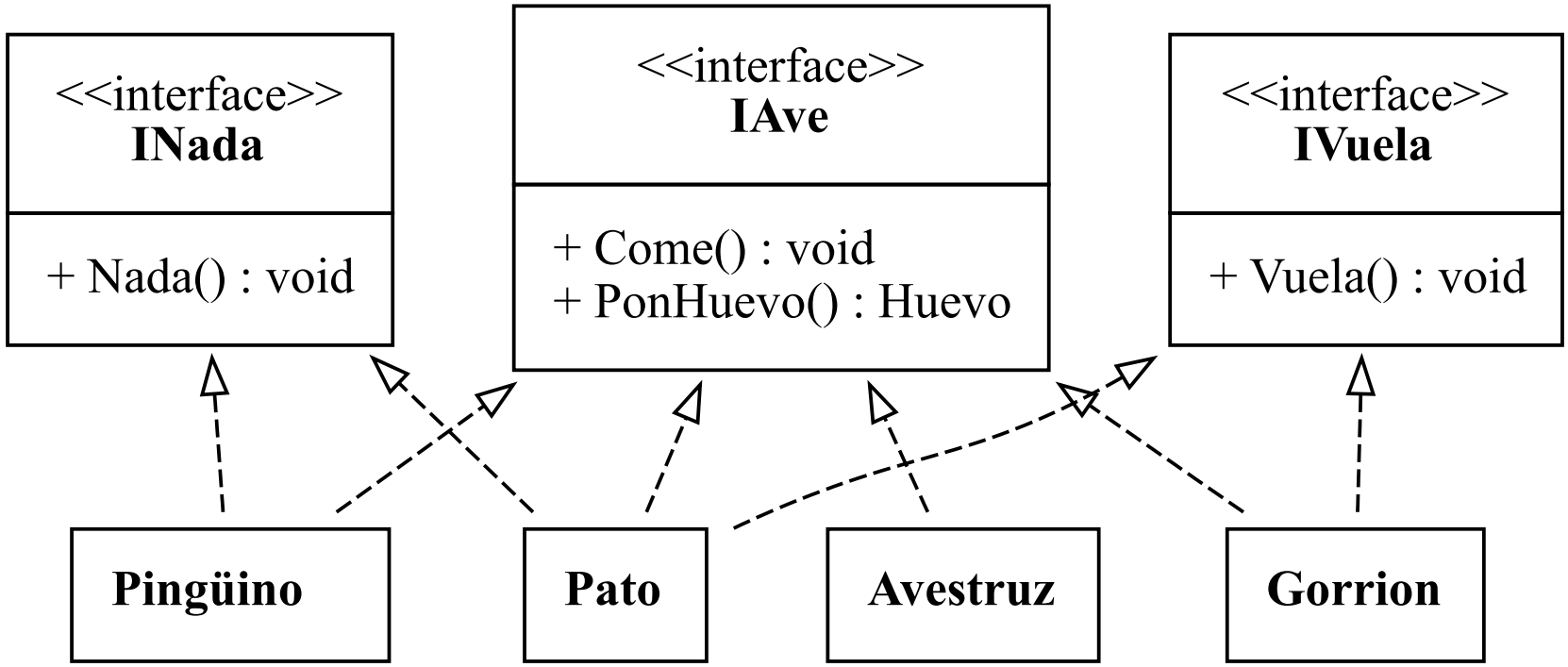
```
class Pato : IAve { Vuela(), Come(), PoneHuevo(), Nada() }

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 genereremos algún tipo de excepción al llamarlas.
- En este caso no estaremos siguiendo el principio de ISP.

PRINCIPIOS S.O.L.I.D.

I – Principio De Segregación De Interfaces (ISP)

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



PRINCIPIOS S.O.L.I.D.

D – 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:
  - Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
  - Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
- En otras palablas podemos decir que, el objetivo de este principio es el uso de interfaces para conseguir que una clase interactue 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.

PRINCIPIOS S.O.L.I.D.

D – Principio De Inversión De Dependencias (DIP)

- 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_ { 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”.

PRINCIPIOS S.O.L.I.D.

D – Principio De Inversión De Dependencias (DIP)

- 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();
    void 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 };
    }
}
```

PRINCIPIOS S.O.L.I.D.

D – Principio De Inversión De Dependencias (DIP)

- Ahora podremos definir Motores que implementen la abstracción.

```
public class MotorGasolina : IMotor {
    public void Acelera() {
        Admision(); Compresion(); Explosion(); Escape();
    }
    public int Revoluciones() { 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 de relacionar las dos clases.

PRINCIPIOS S.O.L.I.D.

D – Principio De Inversión De Dependencias (DIP)

```
public static class VehiculoFactory
{
    public enum TipoMotor { Gasolina, Diesel }

    public static Vehiculo Crea(TipoMotor tipo) {
        Vehiculo v = null;

        switch (tipo) {
            case TipoMotor.MOTOR_DIESEL:
                v = new Vehiculo(new MotorDiesel());
                break;
            case TipoMotor.MOTOR_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.

PRINCIPIOS S.O.L.I.D.

D – Principio De Inversión De Dependencias (DIP)

- 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);
```