Tema 9.2

Descargar estos apuntes

Índice

Profundizando en la Programación Orientada a Objetos

Diferentes Polimorfismos Diferentes Formas

Definiciones de polimorfismo

1. **Polimorfismo de datos o inclusión**
   - **Ya lo hemos visto** cuando estudiamos el concepto de herencia y downcasting. Se basa en el Upcasting o **Principio de Sustitución de Liskov**
   - Es de datos porque, tenemos un objeto o dato con diferentes formas dependiendo del tipo con que lo referenciemos.
   - Además de este, tendremos otras formas de polimorfismo...
2. **Polimorfismo paramétrico o tipos genéricos**
   - Lo vamos a tratar más adelante en este tema 9.
3. **Polimorfismo funcional o sobrecarga**
   - Al igual que el de datos, **ya lo hemos visto** y usado con anterioridad. Pero ahora es cuando lo definiremos formalmente como una característica de los lenguajes OO.
   - Es la **capacidad de definir operaciones o métodos con el mismo identificador** o nombre. Siempre y cuando, la signatura cambie.
   - Recuerda que en C# dos métodos tienen diferente signatura si:
     - Tienen diferente tipo de retorno.
     - Tienen diferente número de parámetros.
     - Teniendo el mismo número de parámetros, algún tipo es diferente.
     - Teniendo el mismo número de parámetros y el mismo tipo alguno tiene el modificador ref o out.

```
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) { ; }
}
```

- **¿Para qué se usa el polimorfismo funcional?**
  - Por ejemplo, lo vímos cuando tuvimos que **definir varios constructores** en una clase.
  - Para **evitar el uso de parámetros opcionales o por defecto** en los métodos.

Parámetros opcionales o por defecto

Una llamada a un método debe proporcionar los argumentos reales para todos los parámetros, sin embargo **se pueden omitir aquellos argumentos** de parámetros opcionales.

✋ Los parámetros opcionales **se definen al final de la lista de parámetros**, después de los parámetros necesarios. Si el autor de la llamada proporciona un argumento para algún parámetro de una sucesión de parámetros opcionales, **debe proporcionar argumentos para todos los parámetros opcionales anteriores** o en su lugar indicar el identificador del parámetro formal.

Veamos esto último a través de un **ejemplo**, de sintaxis:

```
    static class Ejemplo {
        static void Metodo(
                string cadenaRequerida, int enteroRequerido,    // No puedo definir ningún op
                string cadenaOpcional = "", int enteroOpcional = 10)
5               { ... }
        static void Main() {
            Metodo("Cadena obligatoria", 3, "Cadena Opcional", 33); // Correcto
            Metodo("Cadena obligatoria", 3, "Cadena Opcional");     // Correcto enteroOpciona
            Metodo("Cadena obligatoria", 3);                        // Correcto cadenaOpciona
            // Si sabemos el nombre del identificador del parámetro en el método...
            Metodo("Cadena obligatoria", 3, enteroOpcional: 10);    // Correcto cadenaOpciona

13          Metodo("Cadena obligatoria", 3, 10);                    // Incorrecto
            Metodo("Cadena obligatoria", 3, , 10);                  // Incorrecto
        }
    }
```

Se pueden definir en multitud de lenguajes como C#, Python, PHP, Javascript, Kotlin, etc. Sin embargo, **Java no los permite** porque tienen inconvenientes:

- 💀 Mal usados, **pueden dar lugar a baja cohesión** (métodos '*navaja suiza*' o que hacen muchas cosas según los parámetros que le lleguen).
- Ralentizan la ejecución.
- Lleva a confusión a los usuarios de una clase.

> ✋ **Importante:** Por las razones anteriores. **No deberíamos usarlos en métodos públicos**.
> (👁 *fíjate que Microsoft apenas los usa en sus BCL y sí la sobrecarga*)

Caso de estudio:

Vamos a tratar un ejemplo de como evitar parámetros opcionales en los métodos públicos o **en lenguajes que no nos los permitan como Java**, a través de C#.

Si recordamos de temas anteriores, definimos una estructura `Punto2D` que ahora va a tener el método `Desplaza` con el valor del ángulo a 0 de forma opcional.

```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;

    public Punto2D(in double x, in double y) { Y = y; X = x; }

    public Punto2D Desplaza(in double distancia, double anguloGrados = 0D)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        double fila = Y + distancia * Math.Sin(anguloRadianes);
        double columna = X + distancia * Math.Cos(anguloRadianes);
        return new Punto2D(fila, columna);
    }

    public override string ToString() { return $"({X:G2} - {Y:G2})"; }
}
```

Ahora en un programa podríamos instancias un objeto valor Punto2D

`Punto2D p = new Punto2D(2D, 4D);`

y continuación hacer ...

`p.Desplaza(4D);`

Como el parámetro formal `anguloGrados` es opcional, podremos llamar al método `Desplaza` sin especificarlo y en ese caso tomará su valor por defecto `0D` grados, desplazando el punto 4 unidades a la derecha.

🎓 **Caso de estudio (continuación...)**

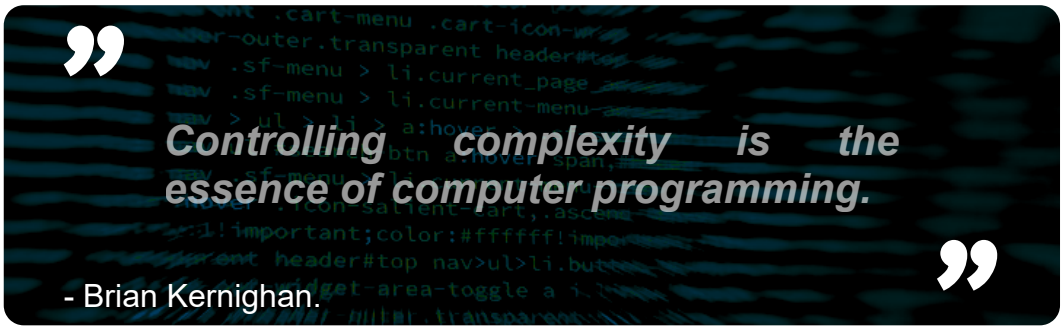
💡 ¿Cómo deberíamos refactorizar el código anterior **usando polimorfismo funcional** o sobrecarga?

La forma más común sería la siguiente...

```
struct Punto2D
{
    // Definimos como privado el método a sobrecargar para no repetir el código
    // además para que no haya conflicto de nombres le ponemos un _
    private Punto2D _Desplaza(in double distancia, double anguloGrados)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        double fila = Y + distancia * Math.Sin(anguloRadianes);
        double columna = X + distancia * Math.Cos(anguloRadianes);
        return new Punto2D(fila, columna);
    }

    // Definimos las sobrecargas públicas, con los parámetros posibles.
    public Punto2D Desplaza(ushort numPosiciones)
    {
        // Aquí decidiremos el valor por defecto.
        return _Desplaza(numPosiciones, 0d);
    }

    public Punto2D Desplaza(ushort numPosiciones, double angulo)
    {
        return _Desplaza(numPosiciones, angulo);
    }
}
```



Propiedades en C#

Una propiedad es una mezcla entre el concepto de campo y el concepto de método. Externamente **es accedida como un campo**, pero internamente es posible **asociar código a ejecutar** en cada asignación o lectura de su valor.

> ✋ Pero la idea principal es que ...*'**es la forma que tiene C# de implementar los métodos accesores y mutadores.**'*

Recordando que era la encapsulación

Antes de hablar de las propiedades **vamos a repasar el concepto de encapsulación** que vimos en temas anteriores y que **va asociado al uso de propiedades**.

📣 Recapitulación de los **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.
- **Disminuir el acoplamiento**, esto es, realizar cambios o actualizaciones en la clases sin preocuparnos cómo están siendo usadas.

📣 Recapitulación de **modificadores de acceso** que hay para clases, tipos, campos y métodos:

- `private` : Accesible solo desde la clase. Es lo que deberíamos poner por defecto a los mutadores y si tenemos alguna duda.
- `public` : Accesible por todos. Puede ser una accesibilidad por defecto para nuestros accesores.
- `protected` : Accesible solo desde la clase o las subclases.
- `internal` : Accesible solo desde clases del artefacto actual.
- `protected internal` **new**: Accesible solo desde clases del artefacto actual y además sean subclases de la clase donde se ha definido.

📣 Recapitulación de cómo las hemos implementado hasta ahora:

- Hemos utilizado la sintaxis de otros lenguajes como **Java** o **C++** para definir los accesores y mutadores que era, básicamente, usar métodos con una notación especial.

```
class Clase
{
    private <Tipo> idCampo;

    private <Tipo> Get<IdCampo>()
    {
            return <idCampo>
    }

    public void Set<idCampo>(<Tipo> <idCampo>)
    {
            this.<idCampo>  = <idCampo>
    }
}
```

Encapsulación a través de propiedades

Como ya hemos comentado, **las propiedades son** un '*azúcar sintáctico*' incluido por el lenguaje C# para abreviar la manera de implementar la encapsulación. De tal manera que la forma básica de implementar los getter y los setter en C# será...

```
class Clase
{
    private <Tipo> idCampo;

    <Tipo> <IdCampo> {
        set
        {
            <idCampo> = value;
        }
        get
        {
            return <idCampo>;
        }
    }
}
```

> 📣 **Nota:** La sintaxis anterior tiene adaptaciones conforme ha ido evolucionando en lenguaje, para **simplificar al máximo su uso**, según casos de uso. Más adelante, una vez tratemos el concepto de propiedad, abordaremos dichas simplificaciones.

Veamos cómo aplicar y usar la nueva sintáxis a traves de la clase `Escritor` que implementamos en el tema 5 y a la que hemos añadido ciertas actualizaciones fáciles de apreciar a primera vista.

A través de ella vamos a ver la forma de usar propiedades y cómo llevar el concepto a las versiones más modernas del lenguaje. Para ello, recordemos pues el código implementando los *Getters* y *Setters* como métodos.

```
class Escritor
{
    private readonly string nombre;
    private readonly DateTime nacimiento;
    private int publicaciones;

    public string GetNombre()
    {
        return nombre;
    }
    public DateTime GetNacimiento()
    {
        return nacimiento;
    }
    public int GetEdad()
    {
        return DateTime.Now.Year - nacimiento.Year;
    }
    public int GetPublicaciones()
    {
        return publicaciones;
    }
    private void SetPublicaciones(in int publicaciones)
    {
        this.publicaciones = publicaciones;
    }

    public Escritor(string nombre, in DateTime nacimiento)
    {
        this.nombre = nombre;
        this.nacimiento = nacimiento;
        SetPublicaciones(0);
    }

    public override string ToString()
    {
        return $"Nombre: {GetNombre()}\n" +
               $"Nacimiento: {GetNacimiento().ToShortDateString()}\n" +
               $"Edad: {GetEdad()}\n" +
               $"Publicaciones: {GetPublicaciones()}";
    }
}
```

La sintaxis equivalente **comentada** usando propiedades sería la siguiente...

```
class Escritor
{
    private readonly string nombre;
    private readonly DateTime nacimiento;
    private int publicaciones;

    // Fíjate que de la línea 9 a la 19 hemos implementado las propiedades
    // solo con el get porque los campos asociados son readonly.
    public string Nombre        // El id de la propiedad debe ser el del campo pero en
    {                           // PascalCasing. Siendo el tipo el mismo que el campo.
        get
        {
            return nombre;      // Como es un getter hacemos un return.
        }
    }
    public DateTime Nacimiento  // Estamos haciendo public lo definido en la
    {                           // propiedad. En este caso únicamente el getter
        get
        {
            return nacimiento;
        }
    }
    public int Edad             // Propiedad calculada a partir de otras.
    {
        get                     // Fíjate que es este getter estamos ya accediendo
        {                       // a la propiedad del campo nacimiento (su get).
            return DateTime.Now.Year - Nacimiento.Year;
        }
    }

    // Dentro de la propiedad Publicaciones definiremos tanto el get como el set.
    public int Publicaciones    // Son public get y set en principio,
    {                           // aunque luego concretaremos el set.
        get
        {
            return publicaciones;
        }
        private set             // Concretamos la accesibilidad general de la
        {                       // propiedad para el set, pero siempre debe ser más
                                // restrictiva que la general para la propiedad.
            publicaciones = value;
        }
    }

    ...
}
```

Del código anterior podemos resumir que:

- Podremos definir solo uno de los dos, `set` o `get` y podrán estar afectados por los modificadores de accesibilidad como el resto de métodos.
- Puedo especificar un modificador de accesibilidad más restrictivo que el de la propiedad a una de las definiciones de `get` y `set`.

Usando las propiedades definidas

Para nosotros sintácticamente será como si estuviéramos accediendo directamente al campo, pero con el nombre en PascalCasing. Sin embargo, se estará ejecutando el código definido en el cuerpo de la propiedad, como sucedía al definir los métodos.

Por ejemplo, si completamos el código de la clase escritor usando la propiedades definidas. Tendremos el siguiente código...

```
class Escritor
{
    // ... código omitido por abreviar

    public Escritor(string nombre, in DateTime nacimiento)
    {
        this.nombre = nombre;
        this.nacimiento = nacimiento;
        Publicaciones = 0;              // Se estará ejecutando el el set de
                                        // la propiedad Publicaciones
    }
    public override string ToString()
    {
        // Estaremos accediendo al get de cada una de las propiedades definidas.
        return $"Nombre: {Nombre}\n" +
               $"Nacimiento: {Nacimiento.ToShortDateString()}\n" +
               $"Edad: {Edad}\n" +
               $"Publicaciones: {Publicaciones}";
    }
}
```

Evolución del uso de propiedades a lo largo del tiempo

Propiedades '*autoimplementadas*'

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 o estemos creando **clases '*ligeras*'**.

Al declarar una propiedad, 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.

Veamos a través de un ejemplo de definición de una clase `Persona` simple, cómo este '*syntactic sugar*' hace que la definición de nuestra clase sea más concisa. Supongamos pues el siguiente código ...

```
class Persona
{
    private readonly string dni;
    private string nombre;

    public string Dni
    {
        get
        {
            return dni;
        }
    }

    public string Nombre
    {
        get
        {
            return nombre;
        }
        private set
        {
            nombre = value;
        }
    }
}
```

el código anterior con las propiedades **autoimplemenradas** pasaría **de 25 a 5 líneas** y tendndría la **misma funcionalidad**...

```
class Persona                   // Estará creando un campo de llamado supongamos _dni
{                               // que es privado y readonly como en el caso de arriba
    public string Dni { get; }  // y del que solo vemos su propiedad Dni.
    public string Nombre { get; private set; }
}
```

Vamos ahora la implementación de nuestra clase escritor usando estas propiedades autoimplementadas.

```
class Escritor
{
    public string Nombre { get; }
    public DateTime Nacimiento { get; }
    public int Publicaciones { get; private set; }

    public int Edad // La propiedad calculada no se puede autoimplementar.
    {
        get
        {
            return DateTime.Now.Year - Nacimiento.Year;
        }
    }

    public Escritor(string nombre, in DateTime nacimiento)
    {
        Nombre = nombre;            // Las propiedades autoimplementadas con solo get
        Nacimiento = nacimiento;    // en el constructor será el único sitio donde se
        Publicaciones = 0;          // puedan asignar.
    }

    public override string ToString()
    {
        return $"Nombre: {Nombre}\n" +
               $"Nacimiento: {Nacimiento.ToShortDateString()}\n" +
               $"Edad: {Edad}\n" +
               $"Publicaciones: {Publicaciones}";
    }
}
```

> ✋ Esta sintáxis será la más concisa, si no vamos ha realizar comprobaciones o transformaciones en los setter y getters.

Miembros con cuerpo de expresión

En el caso de que no usemos propiedades autoimplementadas. Otra forma de realizar un '*syntactic sugar*' al definir las propiedades son los **miembros con cuerpo de expresión** se introdujeron en **C# 6 y 7**. No vamos a profundizar mucho en ellos (usos y significados) hasta el final del curso. Pero son una característica que también podemos encontrar en otros lenguajes como JavaScript, Kotlin, etc.

> 📣 **Definición:** Podemos simplificar diciendo que son aplicables en aquellos métodos que estén formados por una única expresión o instrucción en forma de expresión independientemente de si se evalúan algo o a void.

Veamos un ejemplo con dos métodos donde se cumple la definición anterior.

```
public void SetNombre(string nombre)
{
    Nombre = nombre;
}
public override string ToString()
{
    return $"Dni: {Dni}" + $"Nombre: {Nombre}";
}
```

se pueden representar de una forma más abreviada como la siguiente...

```
public void SetNombre(string nombre) => Nombre = nombre;

public override string ToString() => $"Dni: {Dni}" + $"Nombre: {Nombre}";
```

Si nos fijamos, eliminaremos el cuerpo del método y dejaremos la expresión precedida del operador `=>` y quitando `return` si hubiera.

> **Nota:** A partir de este momento lo iremos usando en los ejemplos para ir quedándonos con este nuevo '*azúcar sintáctico*'.

Veamos cómo sería la sintaxis **si el miembro es una propiedad**. Para ello, vamos a partir de una clase que tiene dos campos y sus respectivas propiedades de acceso que no usan *'cuerpo de expresión'*.

```
class MiClase
{
    private readonly int campo1;
    private string campo2;

    public int Campo1
    {
        get
        {
            return campo1;
        }
    }
    public string Campo2
    {
        get
        {
            return campo2;
        }
        private set
        {
            campo2 = value;
        }
    }
}
```

Si usamos **cuerpos de expresión** en el código anterior quedará...

```
    class MiClase
    {
        private readonly int campo1;
        private string campo2;

6       public int Campo1 => campo1;
7       public string Campo2
        {
            get => campo2;
            private set => campo2 = value;
11      }
    }
```

> ✋ Fíjate que la sintaxis es más simple aún si usamos cuerpos de expresión para la propiedad de un campo de **solo lectura** (**`línea 6`**) que si lo usamos para implementar la propiedad de un campo con su getter y setter (**`líneas 7 a 11`**).

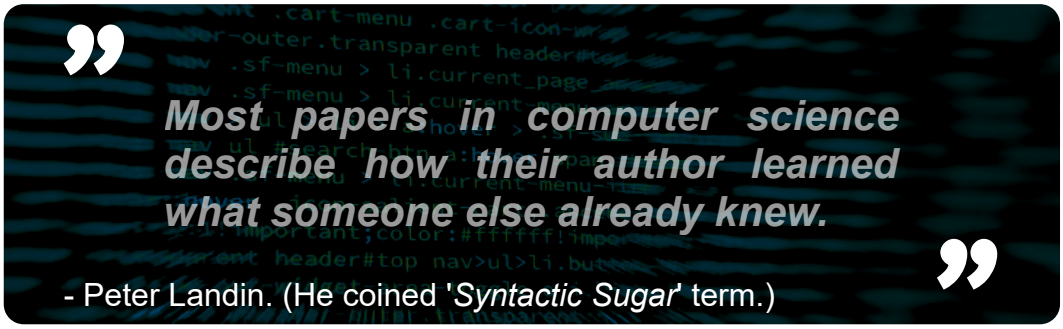
Si refactorizamos la última versión de `Escritor` para usar un cuerpo de expresión en la propiedad `Edad`. Podremos refactorizarla así por **tratarse unicamente de una expresión**.

```
class Escritor
{
    public string Nombre { get; }
    public DateTime Nacimiento { get; }
    public int Publicaciones { get; private set; }

    public int Edad => DateTime.Now.Year - Nacimiento.Year;

    // ... código omitido por abreviar.
}
```

> 📣 **Resumen de casos de uso de propiedades:**
>
> 1. **Autoimplementadas:** No vamos a necesitar conocer el campo ni vamos a añadir ningún control en los mutadores o accesores.
> 2. **Cuerpos de expresión:**
>   Es una propiedad calculada que se puede codificar en una única expresión.
>   Necesitamos un identificador accesible para el campo.
>   Vamos a realizar una trasnformación o comprobación del campo que se puede codificar en una única expresión.
> 3. **'*Tradicionales*':** Además de necesitar acceder al identificador del campo, el cuerpo del getter o setter está formado por más de una instrucción.



Sobrecarga o redefinición de operadores

En los primeros temas 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"`.

Entonces... **¿Podemos cambiar el sentido de la suma según el tipo?**

La respuesta es **sí**. En algunos lenguajes orientados a objetos como **C++**, **C#**, **Python** o **Kotlin** 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**).

> ✋ **Nota:** 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.

Usaremos la siguiente sintaxis general...

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

Vamos a ver cómo aplicar esta sintaxis a través de un ejemplo, que nos mostrará cómo ampliar las capacidades del lenguaje a través de la sobrecarga de operadores.

Supongamos que queremos ampliar nuestros tipos numéricos en el lenguaje, para poder manejar números complejos en su forma binómica.

Para ello podríamos definir un nuevo tipo valor de la siguiente manera ...

> 📣 **Nota:** Aunque nosotros hemos definido la clase como `struct` porque el lenguaje nos lo permite. Si la definición la hiciésemos con una clase (tipo referencia) sería también perféctamente válido.

```
struct Complejo
{
    public double ParteReal { get; }
    public double ParteImaginaria { get; }

    public Complejo(double parteReal, double parteImaginaria)
    {
        ParteReal = parteReal;
        ParteImaginaria = parteImaginaria;
    }

    public override string ToString()
    {
        string texto = $"{ParteReal:G}";
        texto += ParteImaginaria > 0D ? " + " : " - ";
        texto += $"{Math.Abs(ParteImaginaria):G}i";
        return texto;
    }
}
```

Operadores binarios aritméticos

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

```
public static Complejo operator +(Complejo c1, Complejo c2)
{
    double pR = c1.ParteReal + c2.ParteReal;
    double pI = c1.ParteImaginaria + c2.ParteImaginaria;

    return new Complejo(pR, pI);
}
```

> ✋ **Importante:** fíjate que la operación dará como resultado un instáncia nueva del tipo (**`return new Complejo(pR, pI)`**). Esto no sucede solo porque el tipo sea valor. Si estuviéramos implementándolo a través de una clase **sucedería lo mismo**.

Si queremos sumar 2 complejos ahora podremos hacerlo con esta '*extensión del lenguaje*' de forma más intuitiva de la siguiente manera...

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

El resto de operadores binarios aritméticos se implementarían de forma análoga.

Operadores binarios de comparación

Vamos a ver el caso especial de los operadores `==` y `!=`. Estos operadores conjugados deben definirse juntos ya que si definimos uno y no el otro obtendremos un error. Además, al definirlos, también deberemos invalidar los métodos virtuales `Equals` y `GetHashCode` de la clase `Object` puesto que no hacerlo, nos generará un **aviso** del compilador de C# indicándonos que es recomendable hacerlo.

1. Invalidaremos `int GetHashCode()` de tal manera que me devolverá un valor entero que **identifique de forma univoca a los objetos con un determinado estado**. En nuestro caso los números complejos con unos determinados valores de parte real e imaginaria.
   Una forma simple es transformar el objeto en una cadena que identifique el objeto y generar el **Hash** de la cadena ya que la clase `String` invalida este método.
   Podríamos usar el `ToString()` pero es mejor usar una representación como cadena más específica para la comparación. Por ejemplo, para números complejos, la parte real e imaginaria concatenadas con dos decimales.

   ```
   public override int GetHashCode()
   {
       return $"{ParteReal:F2}{ParteImaginaria:F2}".GetHashCode();
   }
   ```

2. Invalidaremos `bool Equals(object obj)` para que me diga si dos números complejos son iguales. Para ello, podemos por ejemplo basarnos en la comparación de su **HashCode** para saberlo.

   ```
   public override bool Equals(object obj)
   {
       return obj is Complejo c && c.GetHashCode() == this.GetHashCode();
   }
   ```

3. Implementamos los operadores `==` y `!=` y re-usamos `Equals` para la comparación.

   ```
   public static bool operator ==(Complejo c1, Complejo c2)
   {
       return c1.Equals(c2);
   }
   public static bool operator !=(Complejo c1, Complejo c2)
   {
       return !c1.Equals(c2);
   }
   ```

Operadores cast

1. Operador de **cast explícito**:

```
// Definición del cast explícito a float
public static explicit operator float(Complejo c)
{
    return Convert.ToSingle(c.ParteReal);
}
// Uso
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.
```

2. Operador de **cast implícito**.

> 💀 **Cuidado:** No implementar pues puede dar lugar a confusión.

```
// Definición del cast implícito a double
public static implicit operator double(Complejo c)
{
    return Convert.ToDouble(c.ParteReal);
}
// Uso
Complejo c = new Complejo(3.7, 2.4);
double d = c; // Asignará 3.7 a d sin darnos error.
```

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 el objeto original que se les pasó como parámetro en lugar del creado en el `return`.

Nota: En ambos casos tras la evaluación, `c` pasará a referenciar al objeto creado en el `return`.

```
// Definición del operador unario ++ (pre y post incremeto)
public static Complejo operator ++ (Complejo c)
{
    return new Complejo(c.ParteReal + 1, c.ParteImaginaria);
}
// Uso
Complejo c = new Complejo(1d, 1d);
// cAux será una copia de c y después c será el nuevo objeto incrementado.
Complejo cAux = c++;
// c será el nuevo objeto incrementado y cAux será una copia del nuevo c.
Complejo cAux = ++c;
```

🎓 **Caso de estudio:**

Vamos a crear las **clases** `Centímetros` y `Metros` para poder hacer operaciones seguras y semánticamente elegantes con ambos tipos de unidades. Para ello, vamos a seguir las siguientes especificaciones.

Ambas poseerán una **propiedad double de solo lectura** denominada `Valor` que almacenará el valor de las unidades que estamos representando.

Vamos a redefinir los siguiente operadores:

1. Operadores **binarios, `+` (suma) y `-` (resta)** que permitan desde ambas clases sumar o restar centímetros con metros y viceversa, permitiendo aplicar por tanto la operación conmutativa de la operación.
   Pare ello, deberemos tener en cuenta que igual que cuando sumamos `int ± double → double`, deberemos decidir a qué se evalúa la suma de `Centímetros ± Metros` y viceversa. En nuestro caso, vamos a evaluar tanto la suma y como la resta a `Centímetros`. Esto es importante porque una misma operación no se puede evaluar a dos tipos diferentes. Por tanto tendremos que.
   - `Metros ± Metros → Metros`
   - `Centímetros ± Centímetros → Centímetros`
   - `Centímetros ± Metros → Centímetros`
   - `Metros ± Centímetros → Centímetros`
2. Operadores **de cast explícitos**:
   - En la clase `Centímetros`: `(Metros)tipoCm;` y `(double)tipoCm;`
   - En la clase `Metros`: `(Centímetros)tipoM;` y `(double)tipoM`;
3. Operadores de comparación `==` y `!=` así como la invalidación de los métodos de `Object` asociados a estos. Ten en cuanta además todas las combinaciones posibles en la comparación: `Metros con Metros`, `Centímetros con Centímetros`, `Metros con Centímetros` y `Centímetros con Metros`.
4. Implementa un pequeño programa de test para probar todos los operadores implementados.

Nota 1: Intenta usar **cuerpos de expresion** en la medida de lo posible y siempre que lo consideres apropiado.

Nota 2: Intenta **reutilizar al máximo el código**. Esto es, usar operadores ya implementados para la implementación de otros, ya que en algún caso el cuerpo de expresión se puede repetir o es análogo.

En las siguientes páginas tienes una propuesta de solución comentada para que puedas compararla con la tuya o echarle un vistazo si te quedas bloqueado/a.

🎓 Caso de estudio (Continuación)

Nota: En esta propuesta de solución hemos usado **cuerpos de expresión**, ya que la mayoría de métodos retornan únicamente una expresión y por tanto es un caso de uso adecuado de los mismos.
👁 En algunos casos para que la implementación del cuerpo de expresión no desbordase por la derecha, se ha bajado la parte del `=> <cuerpo de expresión>;` a la siguiente línea.

```
class Metros
{
    private double Valor { get; }
    public Metros(double metros) { Valor = metros; }

    // Fíjate que este tipo de operadores sencillos es un buen caso
    // de uso adecuado de cuerpos de expresión.
    // Invalidación de ToString para que sepamos mostrarlo con sus unidades.
    public override string ToString() =>  $"{Valor} m";

    // Redefinición de los operadores de cast explícito de Metros a Centímetros y double
    public static explicit operator Centímetros(Metros m)
    => new Centímetros(m.Valor * 100d);
    public static explicit operator double(Metros m) => m.Valor;

    // Redefinición de la suma y la resta binaria de Metros.
    public static Metros operator +(Metros m1, Metros m2)
    => new Metros(m1.Valor + m2.Valor);
    public static Metros operator -(Metros m1, Metros m2)
    => new Metros(m1.Valor - m2.Valor);

    // Redefinición de operadores de comparación (Iguales hasta 2 decimales)
    public override int GetHashCode() => $"{Valor:F2}".GetHashCode();
    public override bool Equals(object obj)
    => obj is Metros m && Math.Abs(Valor - m.Valor) < 1e-2;
    public static bool operator ==(Metros m1, Metros m2) => m1.Equals(m2);
    public static bool operator !=(Metros m1, Metros m2) => ! m1.Equals(m2);

    // Redefinición de la comparación de Metros con Centímetros, re-usamos
    // las implementaciones del cast explícito a Metros y la comparación de
    // Metros con Metros.
    public static bool operator ==(Metros m, Centímetros cm) => m == (Metros)cm;
    public static bool operator !=(Metros m, Centímetros cm) => m != (Metros)cm;

    // Redefinición de la comparación de Centímetros con Metros, re-usamos
    // la implementación de la comparación de Metros con Centímetros.
    public static bool operator ==(Centímetros cm, Metros m) => m == cm;
    public static bool operator !=(Centímetros cm, Metros m) => m != cm;
}
```

🎓 Caso de estudio (Continuación)

Nota: Hemos dejado en esta clase todos los operadores binarios que se evalúan a Centímetros y los de comparación que comparan Centímetros con Centímetros.

```
class Centímetros
{
    private double Valor { get; }
    public Centímetros(double centímetros) {  Valor = centímetros; }

    public override string ToString() =>  $"{Valor} cm";

    // Redefinición de los operadores de cast explícito de Centímetros a Metros y double
    public static explicit operator Metros(Centímetros c) => new Metros(c.Valor / 100d);
    public static explicit operator double(Centímetros c) => c.Valor;

    // Redefinición de la suma y la resta binaria de Centímetros.
    public static Centímetros operator +(Centímetros c1, Centímetros c2)
    => new Centímetros(c1.Valor + c2.Valor);
    public static Centímetros operator -(Centímetros c1, Centímetros c2)
    => new Centímetros(c1.Valor - c2.Valor);

    // Redefinición de la suma y resta binaría de Centímetros con Metros reutilizando
    // el código ya implementado de cast a Centímetros, así como la suma y resta
    // de Centímetros ya implementadas para no repetirnos y rehusar código.
    public static Centímetros operator +(Centímetros c, Metros m)
    => c + (Centímetros)m;
    public static Centímetros operator -(Centímetros c, Metros m)
    => c - (Centímetros)m;

    // Redefinición de la suma y resta binaría de Metros con Centímetros para tener
    // en cuenta así la conmutatividad de la operación.
    public static Centímetros operator +(Metros m, Centímetros c)
    => (Centímetros)m + c;
    public static Centímetros operator -(Metros m, Centímetros c)
    => (Centímetros)m - c;

    // Redefinición de operadores de comparación (Iguales hasta 2 decimales)
    public override int GetHashCode() => $"{Valor:F2}".GetHashCode();
    public override bool Equals(object obj)
    => obj is Centímetros cm && Math.Abs(Valor - cm.Valor) < 1e-2;
    public static bool operator ==(Centímetros c1, Centímetros c2) => c1.Equals(c2);
    public static bool operator !=(Centímetros c1, Centímetros c2) => !c1.Equals(c2);
}
```

Caso de estudio (Continuación)

Vamos a realizar un programa principal para probar la implementación anterior.

```
class CasoDeEstudio
{
    static void Main()
    {
        var c1 = new Centímetros(25);
        var c2 = new Centímetros(200);
        var m = new Metros(2);

        Console.WriteLine($"{c1} + {m} = {c1 + m}");
        Console.WriteLine($"{m} + {c1} = {m + c1}");
        Console.WriteLine($"{m} == {c1} = {m == c1}");
        Console.WriteLine($"{m} == {c2} = {m == c2}");
        Console.WriteLine($"{m} != {c2} = {m != c2}");
        Console.WriteLine($"{m + c1} == {c2 + c1} = {m + c1 == c2 + c1}");

        // De izquierda a derecha...
        //      Suma m con m y se evalúa a m
        //      Suma m con cm y se evalúa a cm
        //      Suma cm con m y se evalúa a cm
        //      Suma cm con cm y se evalúa a cm
        Console.WriteLine($"{m} + {m} + {c1} + {m} + {c2} = {m + m + c1 + m + c2}");

        Console.WriteLine($"{m} = {(Centímetros)m}");
        Console.WriteLine($"{c1} = {(Metros)c1}");

        Console.WriteLine($"{m} = {(double)m}");
        Console.WriteLine($"{c1} = {(double)c1}");
    }
}
```

La salida de ejecutar el programa anterior será la siguiente...

```
25 cm + 2 m = 225 cm
2 m + 25 cm = 225 cm
2 m == 25 cm = False
2 m == 200 cm = True
2 m != 200 cm = False
225 cm == 225 cm = True
2 m + 2 m + 25 cm + 2 m + 200 cm = 825 cm
2 m = 200 cm
25 cm = 0,25 m
2 m = 2
25 cm = 25
```

Caso especial del operador de indización (Indizadores)

Me permiten mapear o indexar datos de un determinado tipo dentro de una clase utilizando el doble corchete `[]` como en las tablas. Es un concepto muy parecido al de las **propiedades** ya que podremos establecer (set) u obtener (get) un valor con el operador.

Una vez definido, este operador lo aplicaremos a los **objetos instanciados** de dicha clase. Además, para indexar **no utilizaremos necesariamente un entero como índice**, sino que también podremos utilizar otros tipos como: **cadenas**, **tipos enumerados**, etc.

Como hemos comentado la sintaxis básica es muy similar a la de las propiedades. Además, también podremos usar la sintáxis de cuerpos de expresión de forma análoga...

```
<tipoDelIndizador> this[<parámetrosQueServiránDeÍndice>]
{
    set
    {
        // Instrucciones para cambiar un "dato" del objeto según el índice o índices.
    }
    get
    {
        // Instrucciones para acceder a un "dato" del objeto según el índice o índices.
    }
}

<tipoDelIndizador> this[<parámetrosQueServiránDeÍndice>]
{
    set => // Expresión para cambiar un "dato". Se evaluará a void.
    get => // Expresión para acceder a un "dato". Se evaluará a <tipoDelIndizador>.
}
```

Un posible uso podría ser encapsular o envolver el acceso a una tabla de datos dentro de una clase.

Por ejemplo, supongamos una clase simplificada `Mago` que contenga su **nombre** y una serie de habilidades adquiridas, de tal manera que cada una estará descrita por un tipo enumerado `Habilidad` definido de forma anidada en `Mago`.
Además, querremos definir un indizador para los objetos `Mago` que me permita acceder o modificar dicha habilidad utilizando como índice el propio enumerado y de esta manera evitar excepciones del tipo IndexOutOfRangeException.

Una posible implementación de la clase `Mago` podría ser ...

```
class Mago
{
    public enum Habilidad { Sabiduría, Poder, Destreza, Energía }
    public string Nombre { get; }
    public ushort[] Valores { get; }
    public Mago(string nombre)
    {
        Nombre = nombre;
        // Dimensionamos el array a los valores del enumerado.
        Valores = new ushort[Enum.GetValues(typeof(Habilidad)).Length];
    }

    // Fíjate que el índice en este caso no será un entero sino el
    // del tipo enumerado de la habilidad.
    public ushort this[Habilidad habilidad]
    {
        get => Valores[(int)habilidad];
        set => Valores[(int)habilidad] = value;
    }

    public override string ToString()
    {
        string texto = $"Habilidades de {Nombre}:\n";
        // Fíjate que con this[habilidad] estamos usando el indizador que
        // acabamos de definir sobre objeto this para acceder al valor de la habilidad.
        foreach(Habilidad habilidad in Enum.GetValues(typeof(Habilidad)))
            texto += $"\t{habilidad} = {this[habilidad]}\n";
        // Nos mostrará siempre las habilidades de forma correcta aunque cambiemos
        // los identificadores en el enumerado.
        return texto;
    }
}
```

Podemos testear nuestra definición de `Mago` con indizador con este sencillo `Main()`...

```
class Program
{
    static void Main()
    {
        var m = new Mago("Gandalf");
        m[Mago.Habilidad.Energía] += 7; // Llamamos al get y después al set
        Console.WriteLine(m);
    }
}
```

Supongamos ahora que queremos usar como índice un texto con el nombre de la habilidad. Podríamos sobrecargar el operador de indización para que aceptase también una cadena con el nombre de la misma. Una posible implementación de esta ampliación sería...

```
    class Mago
    {
        // ... código anterior del ejemplo omitido para abreviar
        // La comprobación del índice lo metemos en este método de clase privado
        // para no repetirlo en e set y el get y así poder una cuerpos de expresión.
        private static Habilidad PasaAEnum(string habilidad)
        {
            if (!Enum.TryParse(habilidad, true, out Habilidad h))
            {
                string m = $"{habilidad} no es un índice válido para Mago.\n"
                + $"Valores posibles: {string.Join(", ", Enum.GetNames(typeof(Habilidad)))}";
                throw new IndexOutOfRangeException(m);
            }
            return h;
        }
16      public ushort this[string habilidad]
        {
            // Podemos usar el indizador de enumerados ya implementado que se encargará
            // ya de hacer la transformación a entero y así no la repetimos.
            get => this[PasaAEnum(habilidad)];
            set => this[PasaAEnum(habilidad)] = value;
22      }
    }
```

Si modificamos ahora el programa de test de nuestra clase de la siguiente manara ...

```
    static void Main()
    {
        var m = new Mago("Gandalf");
        m[Mago.Habilidad.Energía] += 7;
5       m["Poder"] += 2;    // Llamamos al get y después al set del nuevo indizador
        Console.WriteLine(m);
7       m["Fuerza"] += 4;   // Generará una excepción de IndexOutOfRangeException
    }
```

Al ejecutarlo obtendremos la siguiente salida ...

```
Habilidades de Gandalf:
        Sabiduría = 0
        Poder = 2
        Destreza = 0
        Energía = 7

Unhandled exception. System.IndexOutOfRangeException: Fuerza no es un índice válido para Mago.
Valores posibles: Sabiduría, Poder, Destreza, Energía
   at Indizadores.Mago.PasaAEnum(String habilidad) in C:\WD\Program.cs:line 29
   at Indizadores.Mago.get_Item(String habilidad) in C:\WD\Program.cs:line 33
   at Indizadores.Program.Main() in C:\WD\Program.cs:line 54
```

En el ejemplo anterior hemos visto el uso de indizadores para encapsular y controlar el acceso a una tabla dentro de la clase. Pero no siempre es necesario tener siempre una tabla para usar un indizador.

En el siguiente ejemplo podemos indizar un objeto `GYM` de tal manera que dado un día de la semana y un tipo de clase me diga a la hora que se imparte.

> **Nota:** Seguramente habrá **formas más simples y semánticas** de solucionar la propuesta del ejemplo. Pero solo se trata de ver un ejemplo de sintaxis donde definamos un indizador que **use dos índices** y cuyos objetos **no contengan ningún tipo de tabla**. Además de otras características nuevas de C#.

```
class GYM
{
    public enum DiaSemana { Lunes, Martes, Jueves }
    public enum Clase { Yoga, Pilates, GAP }
    // Cómo solo definimos get podemos usar la sintaxis abreviada de
    // cuerpo de expresión análoga a las propiedades, usando un doble switch de expresión
    // En este caso usaremos dos índices.
    public string this[DiaSemana d, Clase c] => d switch
    {
        DiaSemana.Lunes => c switch
        {
            Clase.Pilates => "12:00",
            Clase.GAP => "8:00",
            _ => $"El {d} no hay {c}",
        },
        DiaSemana.Martes => c switch
        {
            Clase.Yoga => "21:00",
            Clase.GAP => "16:00",
            _ => $"El {d} no hay {c}",
        },
        DiaSemana.Jueves => c switch
        {
            Clase.Yoga => "7:00",
            Clase.Pilates => "20:00",
            _ => $"El {d} no hay {c}",
        },
        _ => "Cerrado",
    };
}

var gym = new GYM();
Console.WriteLine(gym[GYM.DiaSemana.Lunes, GYM.Clase.Pilates]);
Console.WriteLine(gym[GYM.DiaSemana.Martes, GYM.Clase.Pilates]);
Console.WriteLine(gym[GYM.DiaSemana.Jueves, GYM.Clase.Pilates]);
```