

Tema 9.3

Descargar estos apuntes en [pdf](#) o [html](#)

Índice

- ▼ [Sobrecarga o redefinición de operadores](#)
 - [Operadores binarios aritméticos](#)
 - [Operadores binarios de comparación](#)
 - [Operadores cast](#)
 - [Operador unario de pre y post incremento/decremento](#)
- ▼ [Anexo de ampliación opcional](#)
 - [Caso especial del operador de indización](#)


Sobrecarga o redefinición de operadores

En los primeros temas ya 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 perfectamente 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 instancia 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 **sucedría 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);
Complejo c3 = c1 + c2; // Sumo de forma 'amigable'.
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.

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

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

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

Operadores cast

1. Operador de **cast explícito**:

```
// Definición del cast explícito a float
public static explicit operator float(Complejo c) => 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) => 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 incremento/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 incremento)
public static Complejo operator ++ (Complejo c)
=> 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.

Para 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 cuenta 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.

💡 Tips:

- Intenta usar **cuerpos de expresión** en la medida de lo posible y siempre que lo consideres apropiado.
- 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.

✚ **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;
}

```

 **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 binaria 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 binaria 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);
}

```

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

```

Anexo de ampliación opcional

Caso especial del operador de indización

El operador de indización, también conocidos como **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 sintaxis 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, queremos 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 el set y el get y así poder usar una expresión.
    private static Habilidad PasaAEnum(string habilidad)
    {
        if (!Enum.TryParse(habilidad, true, out Habilidad h))
        {
            string m = $"{habilidat} no es un índice válido para Mago.\n"
                + $"Valores posibles: {string.Join(", ", Enum.GetNames(typeof(Habilidad)))}";
            throw new IndexOutOfRangeException(m);
        }
        return h;
    }
    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;
    }
}

```

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

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

```

🎓 Caso de estudio:

Vamos a ver un ejemplo, similar al último, que trate de usar todo lo que hemos visto en el tema, repasando conceptos anteriores.

Para ello voy a crear un tipo valor `Hora` que:

- A partir de una cadena del tipo `"HH:MM"` que llegará al constructor y usando expresiones regulares, descompondrá la información para ambos valores en dos propiedades autoimplementadas de solo lectura `HH` y `MM` respectivamente. Generaremos la excepción adecuada si la clase no es la correcta.
- Invalidará `Equals`, `GetHashCode` y `ToString`. Mostrando este último las horas en formato `HH:MM`.
- Redefinirá los operadores `==`, `!=`, `>`, `<`, `>=` y `<=`.

Además, deberemos crear la clase `Horario` que:

- Definirá el **enumerado** `Dia` con los días de la semana de **Lunes** a **Viernes**.
- Definirá un campo de solo lectura `horas` que será un array de objetos `Hora`.
- Definirá un campo de solo lectura `horario` que será una matriz de cadenas (acrónimo de las asignaturas o módulos).
- Un constructor que recibirá el array de horas en el horario y dimensionará la matriz sin datos.
- Definirá un **indizador** que reciba un objeto `Hora` válido dentro del rango del horario de horas y el valor enumerado de día y me devolverá o asignará el valor del módulo impartido. Es importante que si añadimos un valor de hora fuera del horario, generaremos una excepción de `IndexOutOfRangeException`.

📌 **Nota:** si tengo clase de Programación de **8:00** a **8:55** y la hora que me pasan es las 8:22, el indizador devolverá dicho módulo o cambiará el nombre del mismo por otra cadena.

- Definirá un **indizador** que solo permitirá asignar (set) en determinada hora, un array de módulos a impartir durante los diferentes días de la semana.

📌 **Nota:** Este indizador debe reutilizar el indizador que hemos definido anteriormente.

- Invalidaremos el método `ToString` para que muestre el horario por consola.

Por último, vamos a implementar un pequeño programa principal, donde creemos un objeto de clase `Horario` inicializando sus clases con el indizador de hora y posteriormente en un

bucle infinito que se recupere de las posibles excepciones producidas, pediremos día y hora por teclado para mostrar la clase que hay en el horario usando el indizador definido en horario.

Ej. `string actividad = horario[new Hora("9:45"), Dia.Lunes]`

Un ejemplo de ejecución podría ser el siguiente...

Mostraríamos en primer lugar el horario.

	Lunes	Martes	Miércoles	Jueves	Viernes
08:00	BD	BD	BD	BD	BD
08:55	BD	BD	BD	BD	BD
09:50	Prog	SI	SI	SI	Prog
10:45	ED	ED	IT	LM	LM
11:05	Prog	SI	SI	SI	Prog
12:00	Prog	Prog		Prog	Prog
12:55		ED	LM	IT	FOL
13:15		FOL		FOL	SI
14:10		Prog		Prog	SI

Introduce la hora (HH:MM): 8:33

Introduce el día de Lunes a Viernes: Lunes

En ese momento estas en: BD

Introduce la hora (HH:MM): 10:33

Introduce el día de Lunes a Viernes: Lunes

En ese momento estas en: Prog

Introduce la hora (HH:MM): 7:00

Introduce el día de Lunes a Viernes: Martes

Las 07:00 está fuera de horario.

Introduce la hora (HH:MM): 22:00

Introduce el día de Lunes a Viernes: Jueves

Las 22:00 está fuera de horario.

Puedes intentar hacerlo y compararlo con la siguiente propuesta de solución.

En la siguiente página tienes una propuesta de solución ...

Empecemos con la implementación de la clase `Hora`

```

struct Hora
{
    public int HH { get; }
    public int MM { get; }

    public Hora(string hora)
    {
        Regex patron = new Regex(@"^(?<hora>\d{1,2}):(?<minuto>\d{1,2})$");
        Match coincidencia = patron.Match(hora);
        if (!coincidencia.Success)
            throw new ArgumentException(
                $"La hora {hora} no tiene un formato válido HH:MM", "hora");
        HH = ushort.Parse(coincidencia.Groups["hora"].Value);
        MM = ushort.Parse(coincidencia.Groups["minuto"].Value);
    }

    public override string ToString() => $"{HH:D2}:{MM:D2}";
    public override int GetHashCode() => ToString().GetHashCode();
    public override bool Equals(object o) => o is Hora h && h.HH == HH && h.MM == MM;
    public static bool operator ==(Hora h1, Hora h2) => h1.Equals(h2);
    public static bool operator !=(Hora h1, Hora h2) => !h1.Equals(h2);
    public static bool operator >(Hora h1, Hora h2) =>
        (h1.HH > h2.HH) || (h1.HH == h2.HH && h1.MM > h2.MM);
    public static bool operator <(Hora h1, Hora h2) =>
        !(h1 > h2 || h1 == h2);

    public static Hora operator +(Hora h, int minutos)
    {
        int m = h.M + minutos;
        int hora = h.H + m / 60;
        m %= 60;
        return new Hora(hora, m);
    }
}

```

```

class Horario
{
    public enum Dia { Lunes, Martes, Miércoles, Jueves, Viernes }
    // Definimos los campos.
    private readonly string[,] horario;
    private readonly Hora[] horas;

    public Horario(Hora[] horas)
    {
        // Dimensiono la matriz sin clases.
        this.horas = horas;
        horario = new string[horas.Length, Enum.GetNames(typeof(Dia)).Length];
    }

    // Buscaremos el índice que se corresponde en el array de horas
    // (filas en la matriz) que se corresponde con la hora introducida.
    // Por ejemplo, si la primera hora son las 8:00 (ind = 0) y la segunda
    // las 8:55 (ind = 1) y recibo las 8:33 devolveré el índice 0, pero a
    // partir de las 8:55 hasta la siguiente hora devolveré el índice 1.
    private int IndiceSegunHora(Hora hora)
    {
        if (hora == null)
            throw new ArgumentNullException(
                "El indizador no admite valores nulos.", "hora");

        int ind = -1;
        for (int i = 0; i < horas.Length && ind < 0; i++)
        {
            if (hora == horas[i]) ind = i;
            else if (hora < horas[i]) ind = i - 1;
        }

        bool estamosAunEnlaUltimaHora = hora > horas[^1] && hora < horas[^1] + 55;
        if (estamosAunEnlaUltimaHora) ind = horas.Length - 1;
        if (ind < 0)
            throw new ArgumentOutOfRangeException(
                $"La hora {hora} no está fuera del horario.");
        return ind;
    }
}

```

```

public string this[Hora hora, Dia dia]
{
    // Índizador que establece una clase para una Hora y un Día.
    set => horario[IndiceSegunHora(hora), (int)dia] = value;
    // Índizador que obtiene una clase para una Hora y un Día.
    // Si hay un null devolverá HUECO
    get => horario[IndiceSegunHora(hora), (int)dia] ?? "HUECO";
}

public string[] this[Hora hora]
{
    // Indizador que usando el anterior, ponga las clases de Lunes
    // a viernes para una determinada hora.
    set
    {
        int i = 0;
        for (Dia dia = Dia.Lunes; dia <= Dia.Viernes; dia++)
            this[hora, dia] = value[i++];
    }
}

// Mostrar la matriz del horario.
public override string ToString()
{
    const int ANCHO_COLUMNA = 10;
    string[] diasSemana = Enum.GetNames(typeof(Dia));
    StringBuilder texto = new StringBuilder($"{",",-ANCHO_COLUMNA}");
    foreach (string dia in diasSemana)
        texto.Append($"{dia,-ANCHO_COLUMNA}");
    texto.Append("\n");
}

```

```

        for (int i = 0; i < horas.Length; i++)
        {
            texto.Append($"{horas[i],-ANCHO_COLUMNNA}");
            for (int j = 0; j < diasSemana.Length; j++)
                texto.Append($"{horario[i, j],-ANCHO_COLUMNNA}");
            texto.Append("\n");
        }
        return texto.ToString();
    }
}

// Clase con el programa principal
static class Principal
{
    // Método que crea e inicializa un objeto horario.
    static Horario Horario1DAM()
    {
        var horas = new Hora[]
        {
            new Hora("8:00"),    new Hora("8:55"),    new Hora("9:50"),
            new Hora("10:45"),   new Hora("11:05"),   new Hora("12:00"),
            new Hora("12:55"),   new Hora("13:15"),   new Hora("14:10")
        };

        Horario horario = new Horario(horas);
        horario[horas[0]] = new string[] { "BD", "BD", "BD", "BD", "BD" };
        horario[horas[1]] = new string[] { "BD", "BD", "BD", "BD", "BD" };
        horario[horas[2]] = new string[] { "Prog", "SI", "SI", "SI", "Prog" };
        horario[horas[3]] = new string[] { "ED", "ED", "IT", "LM", "LM" };
        horario[horas[4]] = new string[] { "Prog", "SI", "SI", "SI", "Prog" };
        horario[horas[5]] = new string[] { "Prog", "Prog", "", "Prog", "Prog" };
        horario[horas[6]] = new string[] { "", "ED", "LM", "IT", "FOL" };
        horario[horas[7]] = new string[] { "", "FOL", "", "FOL", "SI" };
        horario[horas[8]] = new string[] { "", "Prog", "", "Prog", "SI" };

        return horario;
    }
}

```

```

static void Main()
{
    Horario horario = Horario1DAM();
    while (true)
    {
        try
        {
            Console.WriteLine(horario);
            Console.Write("Introduce la hora (HH:MM): ");
            Hora hora = new Hora(Console.ReadLine());
            Console.Write("Introduce el día de Lunes a Viernes: ");
            Horario.Dia dia = (Horario.Dia)Enum.Parse(
                typeof(Horario.Dia),
                Console.ReadLine());
            Console.WriteLine($"En ese momento estas en: {horario[hora, dia]}");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        Console.ReadKey();
        Console.Clear();
    }
}

```