Tema 5.2

Descargar estos apuntes en pdf o html

Índice

Estructuras de datos básicas

Tablas Multidimensionales

Matrices

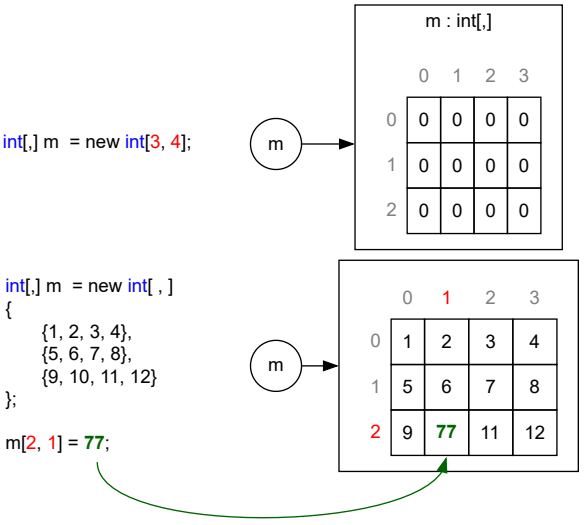
- A las colecciones de 2 dimensiones las denominaremos **matrices**.
- ✋ Intentaremos **evitar** soluciones con **tipos de datos de más de 2 dimensiones**.
  Pues suelen dar lugar a código ofuscado difícil de mantener 💀.

Instanciar objetos de tipo matriz

- Las dimensiones se añaden de **derecha a izquierda** separadas por comas.
  `[…, z, y, x] → […, matriz/profundidad, fila, columna]`
- Podremos definirlas de varias maneras, inicializando a los valores del usuario.

- Pero para mayor claridad, es importante que la definición se trate de separar en varias líneas tabuladas.

```
int[,] m = new int[3, 4]
{
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Puesto que definimos por extensión el compilador
// deduce que las dimensiones son 4 'columnas' y 3 'filas'
int[,] m =
{
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```



Recorrer tablas multidimensionales

- Cada dimensión la recorreremos con un índice.
- Tradicionalmente, desde hace décadas, los programadores han usado la notación índices de la matemática `i`, `j`, `k`. Estos son reconocidos por cualquier programador, ya que es cómo un estándar de facto.
- Por tanto ...
  - Si para los vectores hemos usado `'i'`
  - Para tablas bi-dimensionales (matrices) usaremos `'i'` **fila** y `'j'` **columna**
  - Para tablas tri-dimensionales (cubo), si se diera el caso, usaremos `'i'` **matriz**, `'j'` **fila** y `'k'` **columna**
- Por último, el recorrido se realizará usando bucles anidados que recorren cada índice, siendo el más externo el que recorre `'i'`, después `'j'` y por último **'k'** si lo hubiera.

```
static void Main()
{
    int[,] matriz = new int[,]
    {
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10},
        {11, 12, 13, 14, 15}
    };

    // Recorrido obteniendo longitud por dimensión.
    for (int i = 0; i < matriz.GetLength(0); i++) // Dimensión 0. -> [3,] donde i = fila
    {
        for (int j = 0; j < matriz.GetLength(1); j++) // Dimensión 1 -> [,5] donde j = columna
        {
            Console.Write($"{matriz[i, j],-4}");
        }
        Console.Write("\n\n"); // Salto para cambiar de fila
    }
}
```

Interpretar tablas multidimensionales de 3 dimensiones

No es muy común encontrar este tipo de colecciones, pero nos pueden ayudar a entender un poco más las tablas multidimensionales. Supongamos por ejemplo la siguiente definición.

```
// Trataremos de tabular la inicialización para ganar en claridad.
int[,,] tabla = new int[2, 3, 4]
{
    {
        { 1, 2, 3, 4},
        { 5, 6, 7, 8},
        { 9,10,11,12}
    },
    {
        {13,14,15,16},
        {17,18,19,20},
        {21,22,23,24}
    }
};
```

Si vemos la definición del ejemplo con 3 dimensiones (*'un cubo'*), recordemos que hay que interpretarlas de **derecha a izquierda**.

1. La primera el **4** será: El número de **columnas**.
2. La segunda el **3** será: El número de **filas**.
3. La tercera el **2** será: La profundidad o **el número de matrices de x filas por y columnas**.

> **Nota**
>
> Como ves este tipo de estructuras es muy compleja de entender y de mantener. Por tanto, **no caerán en el examen** y **deberemos evitarlas** en la medida de lo posible.

🎓 **Caso de estudio:**

Somos los administradores de una página de memes y necesitas un programa que te ayude a determinar cuál meme es el más popular. Tenemos una matriz de 2 dimensiones de cadenas donde cada fila representa un meme y contiene:

- **Columna 0**: Nombre del meme (nombre).
- **Columna 1**: Número de "*me gusta*" (texto con valor numérico entero).
- **Columna 2**: Número de "*no me gusta*" (texto con valor numérico entero).

Por ejemplo:

```
string[,] memes = {
    { "Meme del gato", "100", "20" },
    { "Meme del perro", "80", "10" },
    { "Meme del bebé", "120", "50" }
};
```

Crea un programa en C# que:

1. Recorra la matriz y calcule la puntuación de cada meme. La puntuación se calcula restando los "*no me gusta*" de los "*me gusta*".
2. Muestre por pantalla el nombre del meme con la puntuación más alta.

Solución:

```
public class BatallaMemes
{
    public static void Main()
    {
        string[,] memes = {
            { "Meme del gato", "100", "20" },
            { "Meme del perro", "80", "10" },
            { "Meme del bebé", "120", "50" }
        };
        int[] puntuaciones = CalcularPuntuaciones(memes);
        string memeGanador = EncontrarMemeGanador(puntuaciones, memes);
        MostrarMemeGanador(memeGanador);
    }
    static int[] CalcularPuntuaciones(string[,] memes)
    {
        int[] puntuaciones = new int[memes.GetLength(0)];
        for (int i = 0; i < memes.GetLength(0); i++)
        {
            puntuaciones[i] = int.Parse(memes[i, 1]) - int.Parse(memes[i, 2]);
        }
        return puntuaciones;
    }
```

```
    static string EncontrarMemeGanador(int[] puntuaciones, string[,] memes)
    {
        int mayorPuntuacion = puntuaciones[0];
        int indiceGanador = 0;
        for (int i = 1; i < puntuaciones.Length; i++)
        {
            if (puntuaciones[i] > mayorPuntuacion)
            {
                mayorPuntuacion = puntuaciones[i];
                indiceGanador = i;
            }
        }
        return memes[indiceGanador, 0];
    }
    static void MostrarMemeGanador(string meme)
    {
        Console.WriteLine("El meme ganador es: " + meme);
    }
}
```

Combinando colecciones homogéneas

- Podemos hacer que el contenido de una colección homogénea sea otra colección homogénea.
- Deberemos analizar la declaración de **izquierda a derecha** siendo el tipo homogéneo a guardar lo último en tener en cuenta.

```
 3   1    2           1         2         3
int [ ] [ , ] id; // Array de matrices de enteros.
```

  - `char [ , ] [ ] coleccion;` → **Matriz** de **arrays** de **caracteres**.
  - `string [ ] [ ] coleccion;` → **Array** de **arrays** de **cadenas**.
  - `int [ ] [ ] [ , ] coleccion;` → **Array** de **arrays** de **matrices** de **enteros**.
- El caso más común y único que vamos a tratar aquí, son las **tablas dentadas**.

Tablas Dentadas (Jagged Arrays)

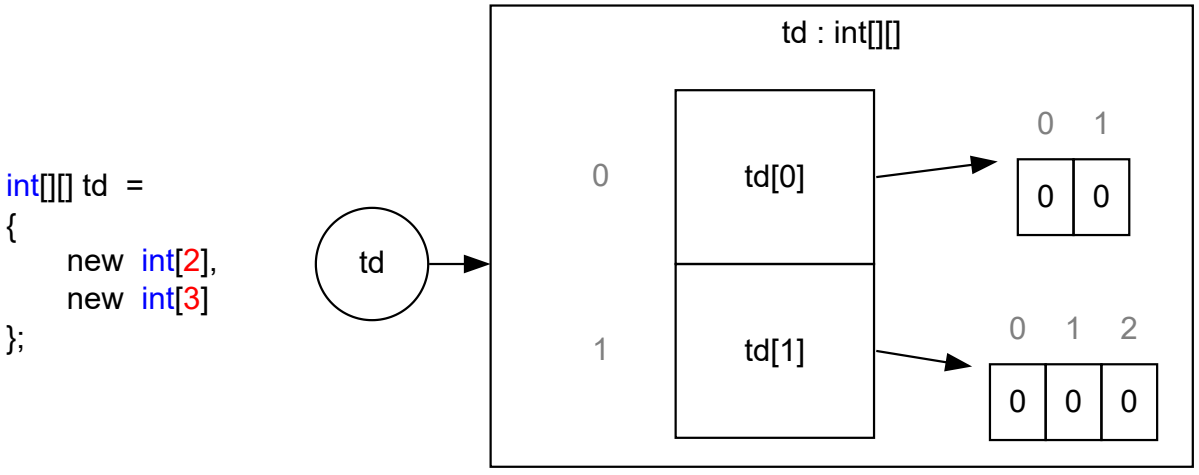
- Es una tabla de tablas o **array de arrays**
- Son estructuras utilizadas cuando necesitamos una matriz, donde la longitud de las filas no va a ser la misma. (De ahí lo de '*dentadas*').

Instanciar tablas dentadas

La sintaxis es la misma que para los arrays solo que los elementos serán objetos array.

- Por ejemplo, para crear una array de array de enteros a los valores por defecto.

```
// Podemos obviar el new int[][] que será deducido por el compilador.
// Se puede interpretar como una matriz donde la primer fila tiene 2 columna y la segunda 3.
int[][] td = new int[][]
{
    new int[2], // [0][0]
    new int[3]  // [0][0][0]
};
```

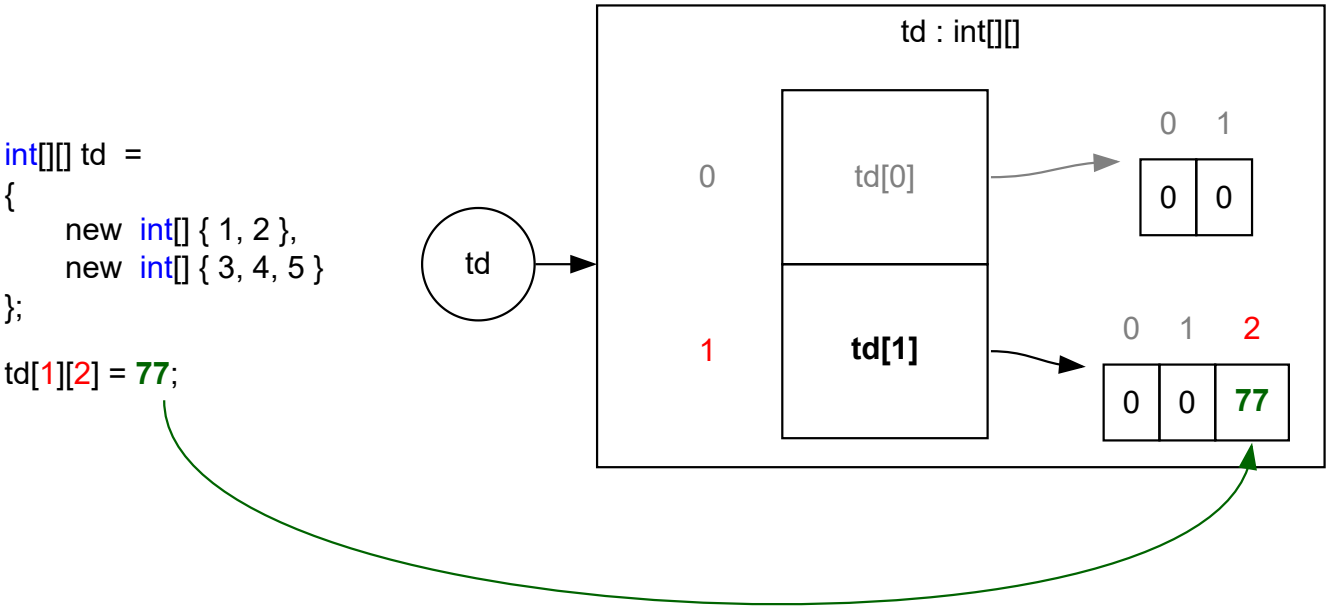


- Si quisiéramos definir por extensión el contenido de la tabla dentada, seguiríamos la sintaxis de definición por extensión de los arrays interiores.

```
int[][] td =
{
    new int[] {1, 2}, ,   // [1][2]
    new int[] {3, 4, 5},  // [3][4][5]
};
```

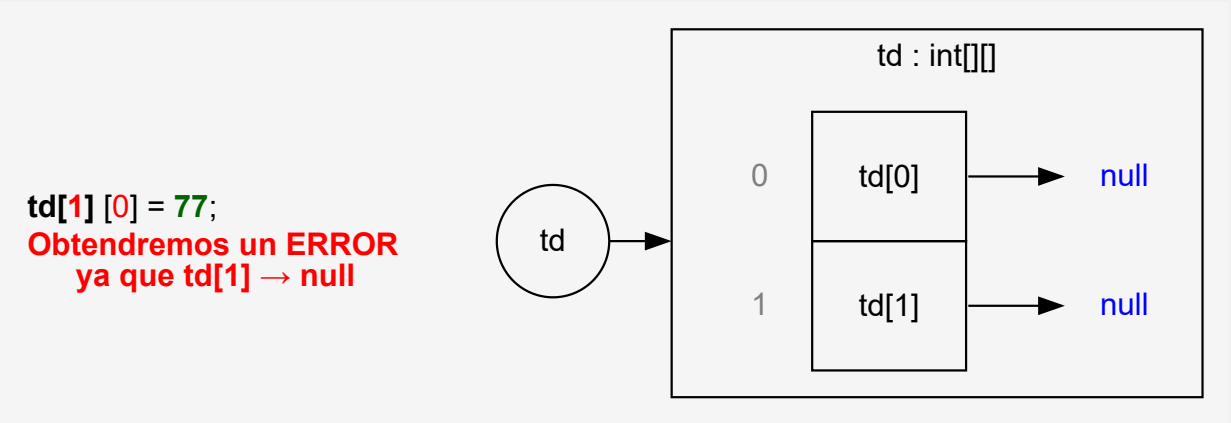
- Para acceder a uno de los datos, primero accederemos a la fila indizando el objeto array que lo contiene. Por ejemplo, si quisieramos cambiar el valor **5** por un **77** accederíamos al array que contiene el 5 a través de `t[1]` (referencia al objeto array que simboliza la segunda fila) y una vez lo tenemos podríamos indizar ya el lugar que ocupa el 5 con `td[1][2]`

```
td[1][2] = 77;

// Sería equivalente ha hacer...

int[] fila2 = td[1];
fila2[2] = 77;
```

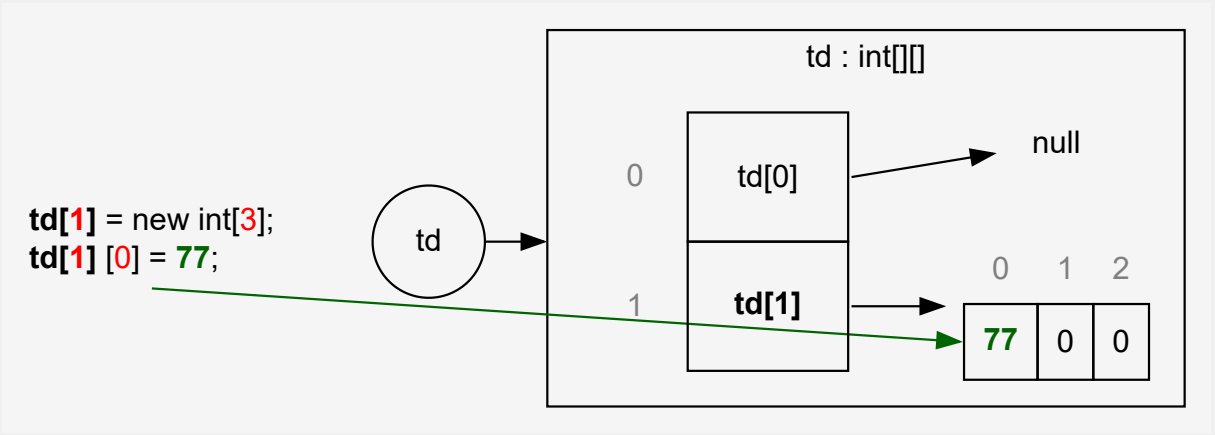


✋ **¿Qué pasa si no instanciamos o no definimos por extensión todos o alguno de los arrays de la tabla dentada?**

Con este código: `int[][] td = new int[2][];` , estamos inicializando un array de **dos arrays** de enteros, pero **sin dimensionar** estos últimos. En ese caso al tratarse de **tipos referencia sin instanciar**, ambos valdrán **null** y no podremos acceder a ellos hasta que los instanciemos dimensionándolos. Por tanto, si hacemos...



Por tanto, para acceder a la posición `td[1][0]` deberemos instanciar primero el array que guardamos en el **índice 1** y posteriormente asignar el valor.



Recorrer tablas dentadas

Lo haremos de forma análoga a como recorremos los arrays.

```
static void Main()
{
    int[][] td =
    {
        new int[] {1, 2},
        new int[] {3, 4, 5, 6, 7, 8},
        new int[] {9, 10, 11}
    };
    // Recorrido con un doble for
    for (int i = 0; i < td.Length; i++)
    {
        for (int j = 0; j < td[i].Length; j++)
            Console.Write($"{td[i][j],-4}");
        Console.Write("\n\n");
    }
    // Recorrido con un doble foreach
    foreach (int[] fila in td)
    {
        foreach (int valor in fila)
            Console.Write($"{valor,-4}");
        Console.Write("\n\n");
    }
}
```

Ejemplo :

Vamos a representar una correspondencia entre comunidades autónomas y sus provincias. De tal manera que, las comunidades irán en un array y en el índice correspondiente en la tabla dentada, irán cada una de las provincias de esa comunidad...

```
[Comuniudad Valenciana] -> 0 [Alicante][Castellón][Valencia]
            [Andalucía] -> 1 [Almería][Cádiz][Córdoba][Granada][Huelva][Jaén][Málaga][Sevilla]
              [Galicia] -> 2 [Lugo][Pontevedra][Orense][La Coruña]
```

Vamos a recorrer ambas estructuras para mostrar el contenido de la siguiente forma:

Recorrido 1

```
Comuniudad Valenciana
        Alicante, Castellón, Valencia
Andalucía
        Almería, Cádiz, Córdoba, Granada, Huelva, Jaén, Málaga, Sevilla
Galicia
        Lugo, Pontevedra, Orense, La Coruña
```

Recorrido 2

```
| Alicante    | Castellón   | Valencia    |
| Almería     | Cádiz       | Córdoba     | Granada     | Huelva      | Jaén        | Málaga      | Sevilla
| Lugo        | Pontevedra  | Orense      | La Coruña   |
```

```
static class Ejemplo
{
    static void Main()
    {
        string[] comunidades = new string[]
        {
            "Comuniudad Valenciana",
            "Andalucía",
            "Galicia"
        };
        string[][] provinciasXComunidades = new string[][]
        {
            new string[] {"Alicante", "Castellón", "Valencia"},
            new string[] {"Almería", "Cádiz", "Córdoba", "Granada", "Huelva", "Jaén", "Málaga", "Sevilla"},
            new string[] {"Lugo", "Pontevedra", "Orense", "La Coruña"}
        };

        // Recorrido 1 : Por 'filas'
        StringBuilder texto = new StringBuilder();
        for (int i = 0; i < provinciasXComunidades.Length; i++)
        {
            texto.AppendLine(comunidades[i]);
            texto.AppendLine($"\t{string.Join(", ", provinciasXComunidades[i])}");
        }
        Console.WriteLine(texto);

        texto.Clear();
        // Recorrido 2 : Elemento a elemento
        for (int i = 0; i < provinciasXComunidades.Length; i++)
        {
            texto.Append("| ");
            for (int j = 0; j < provinciasXComunidades[i].Length; j++)
            {
                texto.Append($"{provinciasXComunidades[i][j], -11}");
                texto.Append(" | ");
            }
            texto.Append("\n");
        }
        Console.WriteLine(texto);
    }
}
```

Enumeraciones

- Internamente se gestionan como objetos de tipo entero. Por tanto, son **tipos valor** y esto significa que en las asignaciones haremos una copia de su valor.
- Son útiles para auto-documentar el código y evitar números mágicos.
- Los utilizaremos siempre que queramos definir un conjunto finito de objetos o estados, **en lugar de definir constantes numéricas**.
- Solo podrán tomar valores, **mútuamente excluyentes**, dentro del rango definido, por lo que nos evitará errores derivados de valores inesperados.
- **Sintaxis:**

```
enum <NombreEnumeración> : <tipoBase>
{
    <Identificadores que definen el conjunto enumerado por extensión>
}
```

  - El identificador del tipo se escribirá en **PascalCasing** y debería estar en singular.
  - Los identificadores de la enumeración se escribirán en **PascalCasing**.
  - El si no lo especificamos por defecto es un **int** aunque podemos especificar otro tipos base enteros como: `byte`, `ushort`, etc.
  - Para acceder a los valores pondremos: `NombreDelEmun.Identificador`
- **Ejemplos:**

```
enum Tamaño
{
    Pequeño, Mediano, Grande
}

Tamaño tamaño = default; // Equivale a hacer tamaño Pequeño
tamaño = Tamaño.Grande;

enum EstadoOrdenador
{
    Encendido, Apagado, Suspendido, Hibernado
}

enum Estación
{
    Primavera, Verano, Otoño, Invierno
}
```

- Si no se especifica valor inicial para cada constante, el compilador les dará por defecto valores que empiecen desde 0 y se incrementen en una unidad para cada constante, según su orden de aparición en la definición de la enumeración. Así, el ejemplo del principio del tema es equivalente a escribir:

```
enum Tamaño : int
{
    Pequeño = 0, Mediano = 1, Grande = 2
}
```

- Es posible modificar el tipo base entero y los valores iniciales de cada constante indicándolos explícitamente, como en el código recién mostrado. Otra posibilidad es alterar el valor base a partir del cual se va calculando el valor de las siguientes constantes, como en este otro ejemplo:

```
enum Tamaño : ushort
{
    Pequeño, Mediano = 5, Grande
}
```

  En este último ejemplo mis enumerados ocuparán menos espacio en memoria por ser entero subyacente `ushort`. El valor asociado a `Pequeño` será **0**, el asociado a `Mediano` será **5**, y el asociado a Grande será 6, ya que como no se le indica explícitamente ningún otro, se considera que este valor es el de la constante anterior más 1.
- Se puede especificarse el valor de un identificador en función del valor de otros como muestra este ejemplo:

```
enum Tamaño
{
    Pequeño, Mediano, Grande = Pequeño + Mediano
}
```

Conversiones con enumeraciones

- `enumerado.ToString()`
  Pasa un enum a cadena.
- `Enum.Parse(Type typeofDelEnum, string id, bool ignoraMayúsculas)`
  `Enum.Parse(Type typeofDelEnum, string id)`
  Pasa una cadena a enum.
- `bool Enum.TryParse(string? id, , bool ignoraMayúsculas, out <MiTipoEnum> valorDelEnum)`
  Intenta asociar una cadena a uno de los id definidos en el enum. Si lo consigue devuelve `true` y el enum a través de `valorDelEnum`.

```
class Ejemplo
{
    enum DiaSemana
    {
        Lunes, Martes, Mircoles, Jueves, Viernes, Sábado, Domingo
    }
    static void Main()
    {
        DiaSemana dia = DiaSemana.Domingo;

        // DE ENUM A CADENA -----------------------------------
        string textoDia = dia.ToString();
        Console.WriteLine(textoDia);

        // DE CADENA A ENUM -----------------------------------
        // Si la cadena no está en el enum se producirá un error
        dia = (DiaSemana)Enum.Parse(typeof(DiaSemana), "Monday", true);
        Console.WriteLine(dia);
        if (Enum.TryParse("Viernes", true, out dia))
            Console.WriteLine(dia);

        // DE ENUM A ENTERO -----------------------------------
        int valorDia = (int)dia;
        Console.WriteLine(valorDia);

        // DE ENTERO A ENUM -----------------------------------
        dia = (DiaSemana)5;
        Console.WriteLine(dia);
    }
}
```

Métodos de utilidad para enumeraciones

- `static Array Enum.GetValues(Type enum)`
  Me devuelve un array del valor enumerado del tipo.
- `static string[] Enum.GetNames(Type enum)`
  Me devuelve un array de cadenas con los valores posibles del enum.
- `static bool Enum.IsDefined(Type enum, object value)`
  Me dice si value está en el enum en alguna de sus formas (enum, int, string).

```
class Ejemplo
{
    enum DiaSemana
    {
        Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo
    }
    static void Main()
    {
        DiaSemana[] diasSemana = (DiaSemana[])Enum.GetValues(typeof(DiaSemana));
        foreach (DiaSemana dia in diasSemana)
            Console.WriteLine($"{dia} = {(int)dia}");

        string[] nombresDiasSemana = Enum.GetNames(typeof(DiaSemana));
        foreach (string dia in nombresDiasSemana)
            Console.WriteLine($"{dia} = {Enum.Parse(typeof(DiaSemana), dia)}");

        Console.WriteLine(Enum.IsDefined(typeof(DiaSemana), "Juernes"));
    }
}
```

Ejemplo:

Implementa un método denominado **PresupuestoAnual**, que devuelva el presupuesto anual en euros, de los diferentes departamentos de una empresa ficticia.
Los posibles departamentos serán **Marketing**, **Compras**, **Ventas**, **RRHH**, **Administración** y su presupuesto será un valor literal de tu elección.
📌 **Nota:** Utiliza una **instrucción switch** para establecer el presupuesto a partir del departamento.

```
class Ejemplo
{
    public enum Departamento
    {
        Marketing, Compras, Ventas, RRHH, Administración
    }

    static double PresupuestoAnual(in Departamento d)
    {
        switch (d)
        {
            case Departamento.Marketing:
                return 30000d;
            case Departamento.Compras:
            case Departamento.Ventas:
                return 40000d;
            case Departamento.RRHH:
                return 10000d;
            case Departamento.Administración:
                return 25000d;
            default:
                // Si en el futuro añadimos un nuevo departamento a nuestra enumeración
                // nos avisará con un error.
                // Nota: El tratamiento de errores lo veremos más adelante.
                throw new NotImplementedException("Falta por tener en cuenta un departamento");
        }
    }
```

```
    static void Main()
    {
        Departamento departamento;
        bool enumCorrecto;
        do
        {
            Console.Write("Departamento: ");
            enumCorrecto = Enum.TryParse(Console.ReadLine(), true, out departamento);
            if (!enumCorrecto)
                Console.WriteLine("Prueba otra vez con " +
                                  $"{string.Join(", ",Enum.GetNames(typeof(Departamento)))}");
        } while (!enumCorrecto);

        Console.WriteLine("El presupuesto anual para " +
                          $"{departamento.ToString().ToLower()} es de " +
                          $"{PresupuestoAnual(departamento)} euros.");
    }
}
```

Tip

Si seleccionamos el '*esqueleto*' de una instrucción `switch(valor)` donde `valor` sea un tipo enumerado. Si vemos las propuestas de refactorización con VSCode (`Ctrl+.`). El editor nos ofrecerá la opción **"*Agregar casos que faltan*"** que añadirá automáticamente todos los case con los valores definidos en la enumeración.

Enumeraciones NO excluyentes (Flags)

- El concepto es muy similar al de flag que vimos con booleanos.
- Es una forma **compacta** y **muy rápida** de guardar varios **flag de estado** asociándolos a un **bit** en memoria en lugar de a una variable booleana.
  Por ejemplo, el valor binario de un byte en memoria puede ser `01100111` y cada bit puede ser un '*flag*' con un significado donde el `1` significa que se cumple y el `0` que no.
- **Para nombrar o identificar** el significado de **los '*flags*' asociados a un bit** utilizaremos una **enumeración**.
- Supongamos la siguiente enumeración **no excluyente** para gestionar los extras en cierto modelo de coche...

```
[Flags]
enum Extra : byte
{
    None              = 0b_0000_0000,   // 0
    Climatizador      = 0b_0000_0001,   // 1
    Navegador         = 0b_0000_0010,   // 2
    FullLed           = 0b_0000_0100,   // 4
    LlantasDeportivas = 0b_0000_1000,   // 8
}
```

Fíjate que hemos añadidos el atributo `[Flags]` sobre la definición de la enumeración para indicar que vamos a definir los nombres de los flags.
Además, hemos hecho que el entero subyacente sea de tipo `byte` y hemos definido por extensión con un literal binario, los valores de cada byte a las potencias de 2 de tal manera que realizará la asociación entre el valor enumerado y el '*flag*' que representa en la byte.
En un principio la byte estará todo a ceros, a trevés de la asignación y para cambiarlo utilizaremos operaciones de bit.

```
Extra extras = default; // default equivale a Extra.None
```

Si queremos añadir uno o varios extras al coche usaremos el **or de bit** * `|`

```
extras |= Extra.Climatizador | Extra.FullLed;
Console.WriteLine(extras); // Mostrará 'Climatizador, FullLed'
Console.WriteLine("{Convert.ToString((byte)extras, 2).PadLeft(8, '0')}"); // Mostrará '00000101'
```

Si queremos ver si hemos establecido algún extra al coche usaremos el **and de bit** `&`
Fíjate que al usar enumerados la operación es mucho más legible.

```
// Hay que tener cuidado con la prioridad de & y por eso ponemos paréntesis.
bool tieneClimatizador = (extras & Extra.Climatizador) == Extra.Climatizador;
Console.WriteLine(tieneClimatizador); // Mostrará 'true'
```

Si queremos quitar algún extra al coche usaremos el **and de bit** `&` y la **negación de bit** `~`

```
extras &= ~Extra.Climatizador;
Console.WriteLine(extras); // Mostrará 'FullLed'
Console.WriteLine("{Convert.ToString((byte)extras, 2).PadLeft(8, '0')}"); // Mostrará '00000100'
```

🎓 **Caso de estudio:**

En el siguiente código vamos a definir una enumeración no excluyente para almacenar los estados combinados de un juego de plataformas. De tal manera que **la primera letra** del '*flag*' me va a **activar** o **desactivar** dicho estado (*El enumerado del estado debería empezar por una letra diferente*). Mostrándome tras cada pulsación como se encuentran los flags, tanto el valor de enumerado como el valor interno en binario del enum.
Además, indicaremos que teclas activarán o desactivarán un deteminado estado.
Nota: Fíjate como el códgo se ha implementado para que funcione, independientemente del nombre que hemos asignado al flag en la enumeración y del número de flags que tengamos definidos.

```
class Ejemplo
{
    [Flags]
    enum PlayerState : byte
    {
        None      = 0b_0000_0000,   // 0
        PowerUp   = 0b_0000_0001,   // 1
        Walking   = 0b_0000_0010,   // 2
        Jumping   = 0b_0000_0100,   // 4
        Attacking = 0b_0000_1000,   // 8
        Shield    = 0b_0001_0000,   // 16
    }

    static string GameOptions()
    {
        StringBuilder options = new StringBuilder("Game keys ( ");
        foreach (string playerState in Enum.GetNames(typeof(PlayerState)))
            options.Append($"'{playerState[0]}' = {playerState} ");
        options.Append(") Press E to Exit.");
        return options.ToString();
    }

    static PlayerState StateAccordingToKey(char key)
    {
        PlayerState stateForKey = PlayerState.None;
        foreach (PlayerState s in (PlayerState[])Enum.GetValues(typeof(PlayerState)))
        {
            if (s.ToString()[0] == key)
            {
                stateForKey = s;
                break;
            }
        }
        return stateForKey;
    }
```

```
    static void Main()
    {
        Console.CursorVisible = false;
        char key;
        PlayerState state = default;
        string gameOptions = GameOptions();

        do
        {
            Console.WriteLine($"PlayerStarte = {state} ({Convert.ToString((byte)state, 2).PadLeft(8, '0')})
            Console.WriteLine(gameOptions);

            key = char.ToUpper(Console.ReadKey(true).KeyChar);

            PlayerState stateToSwitch = StateAccordingToKey(key);
            if ((state & stateToSwitch) == stateToSwitch)
                state &= ~stateToSwitch;
            else
                state |= stateToSwitch;
        }
        while (key != 'E');
    }
}
```