


Tema 5

[Descargar estos apuntes](#)

Índice

1. Índice
2. Programación Orientada a Objetos Básica (POO)
 1. Introducción
 2. Definición de Tipo Abstracto De Datos o TAD
 3. Definición de clase
 1. Elementos que definen una clase
 4. Definición de objeto
 1. Composición de un objeto
 5. Definición de Atributo
 6. Instanciando objetos de una clase
3. Estructuras de datos básicas
 1. Cadenas Inmutables
 1. Introducción
 2. Formas de instanciar objetos cadena
 3. Comparación de cadenas
 4. Obtener la longitud de una cadena
 5. Accediendo a los caracteres de una cadena
 6. Recorrer los caracteres de una cadena
 7. Operaciones de interés sobre objetos cadena
 2. Cadenas Mutables
 1. Instanciar un StringBuilder
 2. Operaciones de interés sobre objetos cadena mutable
 3. Interfaces fluidos o Fluent Interfaces
 3. Colecciones Homogéneas de Tamaño Fijo
 1. Tablas Unidimensionales o (Arrays/Vectores)
 1. Instanciar objetos array
 2. Indexación o acceso a los elementos de un array
 3. Recorrer arrays
 4. Operaciones de interés con arrays
 5.  Pasando y devolviendo arrays en métodos
 2. Tablas Multidimensionales
 1. Matrices
 2. Interpretar tablas multidimensionales de 3 dimensiones
 3. Combinando colecciones homogéneas
 4. Tablas Dentadas (Jagged Arrays)
 1. Instanciar tablas dentadas
 2. Recorrer tablas dentadas
 4. Enumeraciones
 1. Conversiones con enumeraciones
 2. Métodos de utilidad para enumeraciones
 3. Enumeraciones NO excluyentes (Flags)

Programación Orientada a Objetos Básica (POO)

Introducción

- La POO es un paradigma de programación que pretende mejorar aspectos de la programación imperativa tradicional tales como:
 - Abstracción** con la que representamos el problema.
 - Portabilidad** del código y por tanto su reusabilidad.
 - Modularidad** del código y por tanto legibilidad.
- Atacaremos los problemas dividiéndolos en **unidades lógicas denominadas objetos**, que colaborarán entre ellos para resolver el problema.



- Cuando programamos en lenguaje orientado a objetos lo que se debe hacer es atacar los problemas dividiéndolos en **unidades lógicas denominadas objetos**, que colaborarán entre ellos para resolver el problema.

Definición de Tipo Abstracto De Datos o TAD

- Nos preguntaremos que tipos de objetos intervienen en un problema e intentaremos describirlos de forma abstracta, libres de cuestiones de implementación y representación.
- A esta definición abstracta, completa y no ambigua de una estructura de datos junto con el conjunto de operaciones que se pueden hacer sobre ese tipo de datos la denominaremos **TAD**.
- Un TAD puede tener una o más implementaciones.

Definición de clase

- Es una **definición de tipo abstracta**, que permite agrupar datos en una entidad y asociarle un comportamiento.
- Es la implementación total o parcial de un TAD.
- Define objetos que van a tener la misma estructura y comportamiento.
- Añade los conceptos de paso de mensajes, Herencia y Polimorfismo que no se contemplan en los TAD.
- Existen autores que las definen con 2 naturalezas:
 - Como Tipo**: Implementa un TAD con sus atributos y operaciones.
 - Como módulo**: Organización y encapsulación de software.

Elementos que definen una clase

- Un Nombre**: Que describe a la clase.
- Atributos**: Son datos necesarios para describir los objetos creados a partir de la clase.
 - La combinación de sus valores determina el **estado de un objeto**.
- Roles**: Relaciones que una clase establece con otras clases.
- Operaciones, Métodos, Servicios**:
 - Debería ser el único modo de acceder a los atributos.
 - Describe las operaciones posibles sobre un objeto de esa clase.
- Ejemplo definición de una clase para definir una cuenta de banco.

Cuenta
+ Saldo : double + Titular : string
+ Reintegro() : double + Ingreso(in cantidad : double) : void

Definición de objeto

- La creación o **instancia en memoria** de un elemento de la clase.
- De las anteriores definiciones se infiere que: "Un objeto es un conjunto de atributos y métodos que permiten manipular y/o modificar dichos atributos, cambiando así el **estado** del mismo.

Composición de un objeto

- Un **Estado**: que vendrá dado por el valor de sus atributos y su rol durante la ejecución.
- Un **Comportamiento**: que será el modo en que las operaciones cambian a su estado.
- Una **Identidad**: que me permitirá distinguirlo de otros.
 - Dos objetos son iguales si tienen el mismo estado.
 - 👁 No es lo mismo **identidad** que **igualdad**.

cuenta1 : Cuenta	cuenta2 : Cuenta	cuenta3 : Cuenta	cuenta1≡cuenta1 cuenta1=cuenta2 cuenta1≠cuenta3
Saldo = 30000	Saldo = 30000	Saldo = 15000	
Titular = "Xusa"	Titular = "Xusa"	Titular = "Juanjo"	

👉 **Importante:** Para la mayoría de lenguajes actuales como C# **todo** son objetos. De hecho los tipos básicos que hemos visto hasta ahora también son, para él, '*objetos*'.

Definición de Atributo

- También conocidos según el contexto como **Campos** o **Propiedades**.
- Describirá los objetos de una clase y sus valores indicarán el estado de un objeto.
- **¿Qué tipos hay atendiendo a la forma de acceder a ellos?**
 1. **De Instancia**
 - Serán diferentes en cada objeto.
 - Necesitaré de un objeto instanciado en memoria para acceder a ellos.
 2. **De clase**
 - Tendrán el mismo valor en todos los objetos de la clase, por tanto almacenan características comunes a todos ellos.
 - No necesito un objeto instanciado para acceder a ellos.
 - Son visibles desde cualquier método de la clase (ya sea de instancia o no).

Instanciando objetos de una clase

- Para **crear** o **instanciar** objetos de una determinada clase se utiliza el operador **new**, cuya sintaxis es: `new <NombreTipo>(<parametros>)`
- Este operador crea un nuevo objeto del tipo cuyo nombre se le indica. Para ello llama al **constructor** del objeto mas apropiado según los valores que se le pasen en , y devuelve una referencia a la dirección de memoria dinámica donde se ha creado el objeto.

```
Cuenta cuenta1 = new Cuenta();
```

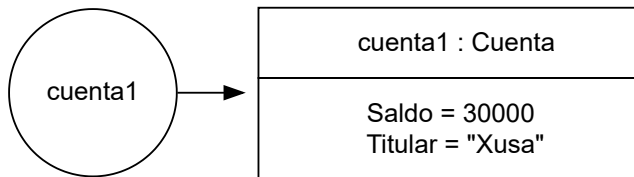
- Para acceder a los atributos y a los métodos de un objeto usaremos el operador .

`<objeto>.<campo>`

`<objeto>.<método>(<parámetros>)`

```
cuenta1.Saldo = 30000;  
cuenta1.Titular = "Xusa";
```

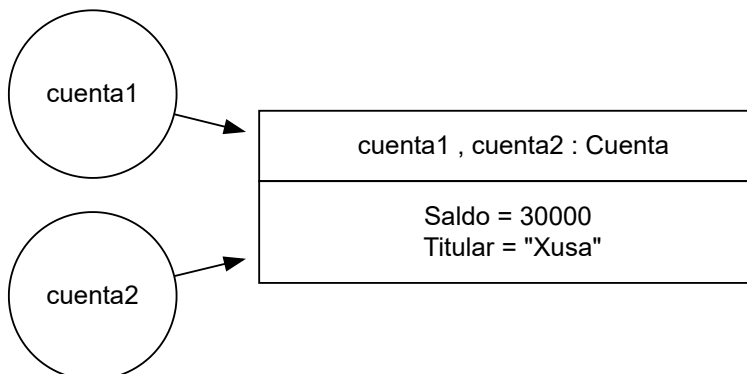
- Los objetos son **tipos referencia**, significa que **cuenta1** será una referencia (como una flecha o puntero) a un objeto en memoria.



- De tal manera que si hacemos ...

```
Cuenta cuenta2 = cuenta1;  
  
Console.WriteLine($"Iguales: {cuenta2 == cuenta1}"); // Mostrará "Iguales: True"
```

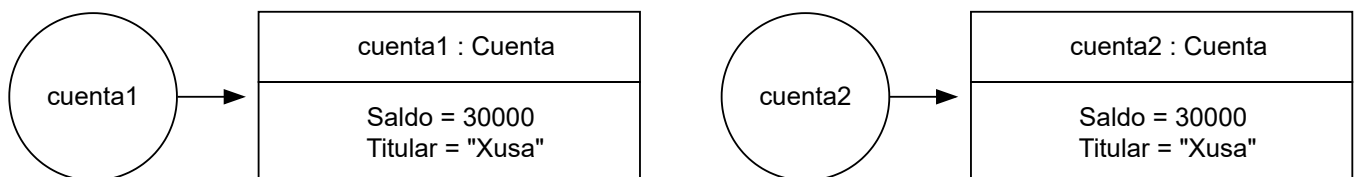
Tendré que ambas referencias 'apuntarán' al mismo objeto en memoria.



- Pero si hacemos una copia del objeto...

```
Cuenta cuenta2 = new Cuenta();  
cuenta2.Saldo = cuenta1.Saldo;  
cuenta2.Titular = cuenta1.Titular;  
  
Console.WriteLine($"Iguales: {cuenta2 == cuenta1}"); // Mostrará "Iguales: False"
```

Ambas referencias 'apuntarán' a diferentes objetos y la comparación `cuenta2 == cuenta1` se evaluará a **False** aunque tengan el mismo contenido



Estructuras de datos básicas

👉 **Importante:** En el **Tema 7** profundizaremos más en la POO: repetiremos estos conceptos, veremos **cómo definir nuestras propias clases** o tipos de datos, además de muchos otros conceptos relacionados con el diseño orientado a objetos.

De momentos a lo largo de los **temas 5 y 6** vamos a ver una serie de clases que podremos encontrar ya definidas en la mayoría de lenguajes actuales, y que me permitirán **instanciar objetos de una serie de estructuras básicas** para manejo de cadenas, arrays, matrices, etc...

Cadenas Inmutables

Introducción

- Las hemos usado ya y se definen a través de la clase `System.String` o su alias `string`.
- Son **tipos referencia**.
- Son **inmutables** porque una vez instanciado un objeto cadena en memoria, no se podrá modificar su contenido.
- La **operaciones** sobre estas cadenas **siempre devolverán un nuevo objeto cadena**.

Formas de instanciar objetos cadena

```
// A partir de un literal de cadena.
string t1 = "Adios";

// A partir de un array de caracteres (Los veremos en breve).
string t2 = new String(new []{ 'H', 'o', 'l', 'a' });

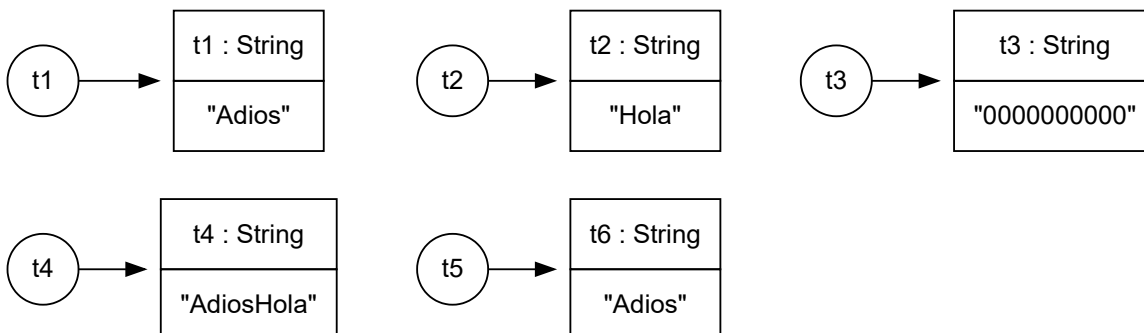
// Inicializando con un carácter de relleno.
string t3 = new String('0', 10);

// Resultado de la concatenación de objetos cadena.
11 // Nótese que la suma de cadenas devuelve un nuevo objeto cadena en memoria. Reflexionaremos sobre esto más adelante.
string t4 = t1 + t2;

// Resultado de una copia del objeto.
string t5 = new String(t1.ToCharArray());

17 // Para hacer copias de cadena NO podremos hacer ...
string t5 = String.Copy(t1); // La función String.Copy no deberíamos usarla pues está marcada como Deprecated.
string t5 = t1; // t1 y t5 serán la misma instancia en memoria.
string t5 = $"{t1}"; // t1 y t5 serán la misma instancia en memoria por optimizaciones de ejecución
string t5 = (string)t1.Clone(); // t1 y t5 serán la misma instancia en memoria.
```

Después de ejecutar este código tendremos **6 objetos cadena instanciados** en memoria, con sus respectivas referencias.



Comparación de cadenas

Puesto que el tipo string es un tipo referencia, en principio la comparación entre objetos de este tipo debería comparar sus direcciones de memoria como acabamos de ver que pasa con cualquier tipo referencia. Sin embargo, si ejecutamos el siguiente código veremos que esto no ocurre en el caso de las cadenas:

```
string t1 = "Adios";
string t2 = t1;
string t3 = new string(t1.ToCharArray());

Console.WriteLine(t1 == t1); // Muestra True
Console.WriteLine(t1 == t2); // Muestra True
7 Console.WriteLine(t1 == t3); // Muestra True, porque aunque sean objetos diferentes la comparación se hace en profundidad.
Console.WriteLine(Object.ReferenceEquals(t1, t3)); // Muestra False
```

Esto se debe a que para hacer más intuitivo el trabajo con cadenas, en C# se ha modificado el operador de igualdad para que cuando se aplique entre cadenas, se considere que sus operandos son iguales sólo si **son lexicográficamente equivalentes** y no si referencian al mismo objeto en memoria. Si se quisiese comparar cadenas por referencia habría que optar por una de estas dos opciones: compararlas con `Object.ReferenceEquals(Object? o1, Object? o2)`.

Obtener la longitud de una cadena

Usaremos la Propiedad `Length` que me indicará cuantos caracteres contiene la cadena.

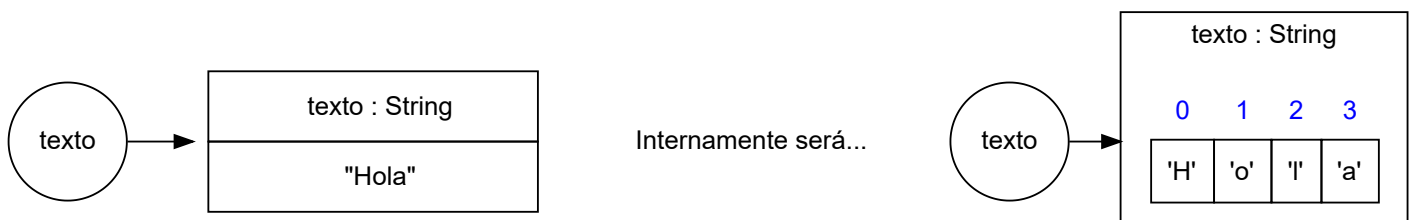
```
Console.WriteLine("Hola".Length); // Mostrará 4
string t = "Adios";
Console.WriteLine(t.Length); // Mostrará 5
```

Accediendo a los caracteres de una cadena

Puedo acceder a un carácter a través del índice, pero **no es posible modificar los caracteres que las forman por ser un objeto inmutable**. Esto se debe a que el compilador comparte en memoria las referencias a literales de cadena lexicográficamente equivalentes, para así ahorrar memoria. Por tanto, si se permitiese modificar los cambios que se hiciesen a través de una variable a una cadena compartida, afectarían al resto de variables que la compartan, lo que podría causar errores difíciles de detectar.

Cuando hacemos...

```
string texto = "Hola";
```

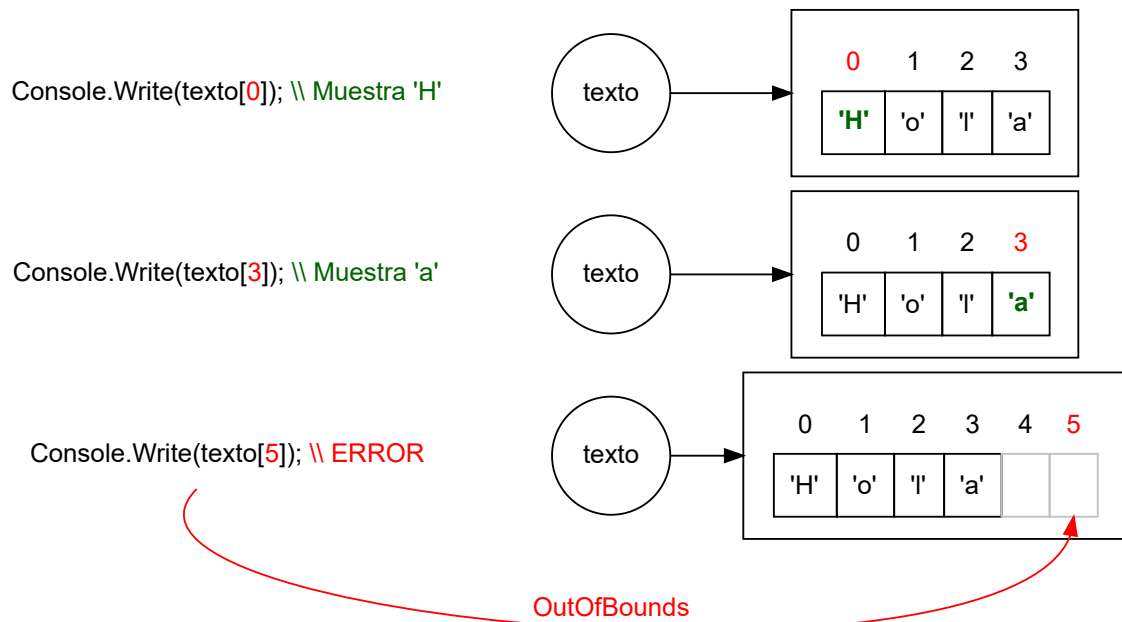


Esto nos permitirá acceder a cada uno de los caracteres que conforman la cadena, a través del operador de indexación `texto[índice]` donde `índice` será un valor entre `0` y `texto.Length - 1`.

```
Console.WriteLine(texto[0]); // Muestra el carácter 'H'
Console.WriteLine(texto[texto.Length - 1]); // Muestra el carácter 'a'

4 // Si sobrepaso el máximo índice posible obtendré un error de OutOfBounds (Fuera de límites)
Console.WriteLine(texto[5]); // ERROR

7 // En las cadenas inmutables no podré modificar el contenido de un carácter.
t[0] = 'M'; // ERROR
t[texto.Length - 1] = 'i'; // ERROR
```



Recorrer los caracteres de una cadena

Básicamente podremos hacerlo a través de un **bucle for** y un **bucle foreach**

```
// Recorrido con for
2 for (int i = 0; i < s.Length; i++)
{
    char c = s[i];
    Console.WriteLine(c);
}
```


La instrucción **foreach** es una variante del **for** pensada, especialmente, para compactar la escritura de códigos donde se realice algún tratamiento a todos los elementos de una colección.

- Tendremos un bucle con tantas iteraciones como elementos en el array.
- En cada iteración la variable definida, que será local al ámbito del foreach, tomará el valor de cada uno de los elementos del array, de forma ordenada.

```
// Recorrido con foreach
// El bucle foreach es más apropiado en caso de no necesitar
// saber en que posición está el caracter.
4 foreach (char c in s)
{
    Console.WriteLine(c);
}

// En ambos caso en la variable c tendremos cada uno de los caracteres de la cadena.
```

Operaciones de interés sobre objetos cadena

- Podré realizar numerosas operaciones sobre los objetos cadena.
-  **Importante:** Recuerda que al ser inmutables, si modifican el contenido de la cadena me devolverán un nuevo objetos cadena.
- Para ello dispondré de [numerosos métodos ya definidos](#) en la clase.
 - **Trimming:** Recortar caracteres especiales a izquierda y/o derecha.
 - **Padding:** Añadir caracteres de relleno a izquierda y/o derecha.
 - etc.

Veamos algunos de los más útiles a través de ejemplos...

- `cadena.ToUpper()` / `cadena.ToLower()`

Devuelve un nuevo objeto cadena con equivalente en mayúsculas de la cadena actual.

```
string cadena = "Hola";
Console.WriteLine(cadena.ToLower()); // Muestra "hola"
```

Ejemplo 1:

Define un método con la signatura `static string Capitaliza(string s)` que recibe una cadena y me devuelve la otra capitalizada esto es. Si entra la cadena `"esto es un título capitalizado"` me devolverá `"Esto Es Un Título Capitalizado"`

```
class Program
{
    static string Capitaliza(string s)
    {
        string sCapitalizada;


        if (!string.IsNullOrEmpty(s)) // Si la cadena no es null o vacía.
        {
            // Pasamos el primer carácter a cadena y después esa cadena a mayúscula.
            sCapitalizada = s[0].ToString().ToUpper();

            // Otra forma de hacerlo es pasar primero el carácter a mayúscula y después a cadena.
            // sCapitalizada = char.ToUpper(s[0]).ToString();

            // Recorremos el resto de caracteres de la cadena
            // Si el anterior es un espacio es blanco lo concatenamos a la cadena capitalizada (pasándolo previamente a cadena)
            // sino es espacio en blanco lo dejamos tal cual (pasándolo previamente a cadena)
            // Usamos un bucle for en lugar de un foreach porque necesitamos saber la posición anterior.
            for (int i = 1; i < s.Length; i++)
                sCapitalizada += char.IsWhiteSpace(s[i - 1]) ? s[i].ToString().ToUpper() : s[i].ToString();
        }
        else
            sCapitalizada = s;

        return sCapitalizada;
    }

    public static void Main()
    {
        Console.WriteLine(Capitaliza("hola caracola")); // Mostrará "Hola Caracola"
    }
}
```

 **Importante:** Fijémonos que cada vez que hacemos `ToString()` y **concatenamos** esa cadena a la cadena capitalizada. Estamos creando objetos cadena nuevos en memoria que luego se van a desechar. Esto es, **para una cadena de 100 caracteres crearemos 200 objetos cadena**, lo cual **es muy costoso** a nivel de proceso.

Por esta razón **este tipo de procesos los realizaremos con cadenas mutables** que veremos más adelante. Es más, analizadores semánticos de código como SonarLint me indicarán que este tipo de concatenaciones no es una buena práctica.

Ejemplo 2:

Escribe un método que **"Normalize para comparación"** una cadena de entrada. Consideraremos normalizar quitar tildes, diéresis y pasar todos los caracteres a minúsculas.

```
static string Normaliza(string s)
{
    string sN = "";
    foreach (char c in s)
    {
        sN += char.ToLower(c) switch
        {
            'á' => "a",
            'é' => "e",
            'í' => "i",
            'ó' => "o",
            'ú' => "u",
            'ü' => "u",
            _ => c.ToString()
        };
    }
    return sN;
}
```

- **cadena.ToCharArray()**

Convierte una cadena de caracteres en una array de caracteres.

En el [esquema](#) de arriba. Hemos visto que internamente al objeto cadena y transparente para nosotros, la cadena se guarda dentro del objeto como un array, por lo que esta conversión es poco costosa.

Más adelante veremos la utilidad de esta conversión.

```
string cadena = "Hola";
char[] array = cadena.ToCharArray();
```

- **Cadena.IndexOf(char)**

Cadena.IndexOf(string)

Devuelve la posición de la primera ocurrencia del carácter o la cadena empezando a buscar por índice hasta Length (0 si no se lo paso) y si no encuentra ninguna ocurrencia retorna -1.

```
string nombre = "Manuel García";
Console.WriteLine(nombre.IndexOf('a')); // Imprime : 1
Console.WriteLine(nombre.IndexOf("el")); // Imprime : 4
Console.WriteLine(nombre.IndexOf("a", 3)); // Imprime : 8
Console.WriteLine(nombre.IndexOf("z")); // Imprime : -1
```

- **Cadena.LastIndexOf(...)**

Devuelve la posición de la última ocurrencia del carácter o la cadena empezando a buscar por índice hasta 0 (Length si no se lo paso) y si no encuentra ninguna ocurrencia retorna -1.

```
string nombre = "Manuel García";
Console.WriteLine(nombre.LastIndexOf("a")); // Imprime : 12
Console.WriteLine(nombre.LastIndexOf('a', 2)); // Imprime : 1
```

Ejemplo 1:

Escribe un método, que **sin recorrer la cadena**, devuelva un entero indicando el número de apariciones de una palabra en una frase. Llámalo: `int Coincidencias(string palabra, string frase)`

```
static class Program
{
    static int Coincidencias(string palabra, string frase)
    {
        int veces = 0;
        int iComienzo = -1;
        while ((iComienzo = frase.IndexOf(palabra, iComienzo + 1)) >= 0) veces++;
        return veces;
    }

    static void Main()
    {
        string s = "oca, gallina, perro, perro, oca, oca, cerdo";
        Console.WriteLine(Coincidencias("oca", s)); // Muestra 3
        Console.WriteLine(Coincidencias("perro", s)); // Muestra 2
        Console.WriteLine(Coincidencias("caballo", s)); // Muestra 0
    }
}
```

Ejemplo 2:

Escribe una función que reciba dos palabras, y averigüe si son anagramas (están formadas por las mismas letras pero en diferentes posiciones).

Ej. (aprobar y probará), (códigos y cogidos), etc.

```
// Una aproximación simplificada podría ser.
static string Normaliza(string s)
{
    const string TILDES = "áéíóúü";
    const string SUSTITUCIONES = "aeiouu";
    string sN = "";
    foreach (char c in s)
    {
        int i = TILDES.IndexOf(c);
        sN += i >= 0 ? SUSTITUCIONES[i].ToString() : c.ToString();
    }
    return sN;
}

static bool Anagramas(string p1, string p2)
{
    bool anagramas = (p1.Length == p2.Length);
    p1 = Normaliza(p1);
    p2 = Normaliza(p2);
    if (anagramas)
    {
        for (int i = 0; i < p1.Length && anagramas; i++)
            anagramas = p1.IndexOf(p2[i]) > -1 && p2.IndexOf(p1[i]) > -1;
    }
    return anagramas;
}
```

Ejemplo 3:

Haz un método que, a partir de una contraseña de entrada, indique el nivel de protección, devolviendo una de las siguientes cadenas:

- **Muy débil:** Contiene solo números y tiene menos de 8 caracteres.
- **Débil:** Contiene solo letras y tiene menos de 8 caracteres.
- **Fuerte:** Contiene letras y números y al menos 8 caracteres.
- **Muy fuerte:** Contiene letras y/o números, caracteres especiales y al menos 8 caracteres.

```
static class Program
{
    static bool SoloNumeros(string texto)
    {
        bool todoEsLoQueBusco = true;
        for (int i = 0; i < texto.Length && todoEsLoQueBusco; i++)
            todoEsLoQueBusco = char.IsDigit(texto[i]);
        return todoEsLoQueBusco;
    }

    static bool SoloLetras(string texto)
    {
        bool todoEsLoQueBusco = true;
        for (int i = 0; i < texto.Length && todoEsLoQueBusco; i++)
            todoEsLoQueBusco = char.IsLetter(texto[i]);
        return todoEsLoQueBusco;
    }

    static bool HayUnCaracterEspecial(string texto)
    {
        bool especial = false;
        for (int i = 0; i < texto.Length && !especial; i++)
            especial = !char.IsLetterOrDigit(texto[i]);
        return especial;
    }

    static string NivelSeguridad(string clave)
    {
        string nivel;

        if (clave.Length < 8 || SoloNumeros(clave))
            nivel = "Muy Débil";
        else if (SoloLetras(clave))
            nivel = "Débil";
        else if (!HayUnCaracterEspecial(clave))
            nivel = "Fuerte";
        else
            nivel = "Muy Fuerte";

        return nivel;
    }

    static void Main()
    {
        Console.Write("Introduce una clave: ");
        string clave = Console.ReadLine();
        Console.WriteLine($"Su nivel de seguridad es {NivelSeguridad(clave)}");
    }
}
```

- **Cadena.Substring(int startIndex, int length)**

Cadena.Substring(int startIndex)

Recupera una subcadena de la instancia. La subcadena comienza en una posición de carácter especificada y tiene una longitud especificada.

```
string s1 = "aaBBBaa";  
// Extraigo una subcadena de 3 caracteres desde el índice 2.  
string s2 = s1.Substring(2, 3);  
Console.WriteLine(s2); // Muestra "BBB"
```

- **Cadena.Remove(int startIndex, int length)**

Cadena.Remove(int startIndex)

Devuelve una nueva cadena en la que se ha eliminado un número de caracteres especificado en la instancia actual a partir de una posición especificada.

```
string s1 = "Estofado";  
// Elimino una subcadena de 3 caracteres desde el índice 2.  
string s2 = s1.Remove(3, 2);  
Console.WriteLine(s2); // Muestra "Estado"
```

- **Cadena.Replace(char oldChar, char newChar)**

Cadena.Replace(string oldValue, string newValue)

Devuelve una nueva cadena en la que todas las apariciones de una cadena especificada en la instancia actual se reemplazan por otra cadena especificada.

```
string s1 = "oca, gallina, perro, perro, oca, gallina";  
// Sustituimos "oca" por "pato"  
s1 = s1.Replace("oca", "pato");  
Console.WriteLine(s1); // Muestra "pato, gallina, perro, perro, pato, gallina"
```

Cadenas Mutables

Si deseamos utilizar cadenas mutables. Debemos usar la clase `System.Text.StringBuilder`, que funciona de manera similar a `string`, pero permite la modificación de sus cadenas en tanto que estas no se comparten en memoria.

🔔 **Nota:** Las operaciones de acceso a un carácter son iguales que en los objetos de la clase `string`. Sin embargo no será posible recorrerlas con un `foreach`

Instanciar un StringBuilder

- Para crear objetos de este tipo, basta pasar como parámetro de su constructor el objeto `string` que contiene la cadena a representar mediante un `StringBuilder`, y para convertir un `StringBuilder` en `String` siempre puede usarse su método `cadenaMutable.ToString()`.

```
string s = "Hola";
StringBuilder sb = new StringBuilder(s); // Pasamos de String a StringBuilder
sb[sb.Length-1] = 'i';                  // Podemos modificar un caracter por ser mutable
s = sb.ToString();                      // Pasamos de StringBuilder a String
Console.WriteLine(s);                   // Muestra "Holi"
```

Ejemplo 1:

Reimplementemos el método con la signatura `static string Capitaliza(string s)` que recibe una cadena y me devuelve la otra capitalizada esto es. Si entra la cadena `"esto es un título capitalizado"` me devolverá `"Esto Es Un Título Capitalizado"`

Nota: Este es un caso claro de uso de `StringBuilder` ya que hará que el método sea muchísimo más eficiente.

```
static class Program
{
    static string Capitaliza(string s)
    {
        string sCapitalizada;
        if (!string.IsNullOrEmpty(s))
        {
            // Transformamos en una cadena mutable si hay al menos 1 carácter a capitalizar.
            9  StringBuilder sb = new StringBuilder(s);
            // Cambiamos el nuevo objeto cadena mutable las veces que queramos.
            sb[0] = char.ToUpper(sb[0]);
            for (int i = 1; i < s.Length; i++)
            {
                sb[i] = char.IsWhiteSpace(sb[i - 1]) ? char.ToUpper(sb[i]) : sb[i];
            }
            // Al finalizar volvemos a transformarlo a cadena inmutable.
            15 sCapitalizada = sb.ToString();
        }
        else
        {
            sCapitalizada = s;
            return sCapitalizada;
        }

        public static void Main()
        {
            Console.WriteLine(Capitaliza("hola caracola")); // Mostrará "Hola Caracola"
        }
    }
}
```

Ejemplo 2:

Lo mismo sucederá con el método de normalización que estaría más correcto con la siguiente implementación.

```
static string Normaliza(string s)
{
    string sNormalizada;
    if (!string.IsNullOrEmpty(s))
    {
        const string TILDES = "áéíóúü";
        const string SUSTITUCIONES = "aeiouu";
        StringBuilder sb = new StringBuilder(s);
        foreach (char c in s)
        {
            int i = TILDES.IndexOf(c);
            sb[i] = i >= 0 ? SUSTITUCIONES[i] : c;
        }
        sNormalizada = sb.ToString();
    }
    else
        sNormalizada = s;
    return sNormalizada;
}
```

Operaciones de interés sobre objetos cadena mutable

- 👉 **Importante:** Se crean dimensionándose a una **capacidad** de caracteres por defecto aunque la cadena esté vacía y por tanto su **longitud** cero. Además, si en vez de modificar un carácter ya existente, queremos añadir algo al final de la cadena usaremos el método `cadenaMutable.Append(...)`.

```
// Aunque la cadena de entrada mide 4 reserva espacio o capacidad en el StringBuilder para 16 caracteres.
StringBuilder sb = new StringBuilder("Hola");
// Podré modificar el primer carácter sin problemas.
sb[0] = 'M';

Console.WriteLine($"Longitud = {sb.Length} Capacidad = {sb.Capacity}"); // Muestra Longitud = 4 Capacidad = 16

// Por defecto reserva espacio para 16 caracteres.
sb = new StringBuilder();

// sb[0] = 'M'; Daría ERROR porque la cadena está vacía.

// Pero si añado un carácter, su sb.Longitud es ahora 1 y como teníamos capacidad de 16 la operación tendrá bajo coste
sb.Append('M');
// Podré modificar ahora posiciones por debajo de sb.Length-1 aunque la capacidad sea superior.
sb[0] = 'I';

Console.WriteLine($"Longitud = {sb.Length} Capacidad = {sb.Capacity}"); // Muestra Longitud = 1 Capacidad = 16

// Reservamos un espacio previo para 2 caracteres.
sb = new StringBuilder(2);
Console.WriteLine($"Longitud = {sb.Length} Capacidad = {sb.Capacity}"); // Muestra Longitud = 0 Capacidad = 2

// sb[0] = 'M'; Daría ERROR porque la cadena está vacía.

sb.Append('1');
sb.Append('2');
// Como ya no queda espacio reservado o capacidad para el nuevo carácter.
// Aumenta la capacidad automáticamente en un valor no controlado por nosotros, pero tendrá un coste mayor que si la tuviéramos pre
sb.Append('3');

Console.WriteLine($"Longitud = {sb.Length} Capacidad = {sb.Capacity}"); // Longitud = 3 Capacidad = 4
```

- **StringBuilder cadenaMutable.Insert(int indice, <elemento>)**

Inserta elementos en una cadena mutable.

```
StringBuilder cadenaMutable = new StringBuilder("aaaa");
string cadena = "bbbb";
// Inserto la cadena después del índice 2
4 cadenaMutable.Insert(2, cadena);
Console.WriteLine(cadenaMutable); // Muestra: aabbbbaa

char[] array = new char[] { 'C', 'D' };
// Inserto el array después del índice 2
9 cadenaMutable.Insert(2, array);
Console.WriteLine(cadenaMutable); // Muestra: aaCDbbbaa
```

- **StringBuilder cadenaMutable.Remove(int indice, int longitud)**

Elimina n caracteres a partir de un índice.

```
StringBuilder cadenaMutable = new StringBuilder("aBBa");
// Elimino 2 caracteres a partir del índice 1
cadenaMutable.Remove(1, 2);
Console.WriteLine(cadenaMutable); // Muestra: aa
```

- **StringBuilder cadenaMutable.Replace(<elementoAReemplazar> , <elementoDeReemplazo>)**

Reemplazo ocurrencias de un **carácter** por otro o de una **cadena** por otra.

```
StringBuilder cadenaMutable = new StringBuilder("Banana");
// Reemplazo ocurrencias del carácter 'a' por 'e'
3 cadenaMutable.Replace('a', 'e');
Console.WriteLine(cadenaMutable); // Muestra: Benene

cadenaMutable = new StringBuilder("Dile a Juanjo que lo entiendo");
// Reemplazo ocurrencias de la cadena "Juanjo" por "Xusa"
8 cadenaMutable.Replace("Juanjo", "Xusa");
Console.WriteLine(cadenaMutable); // Muestra: "Dile a Xusa que lo entiendo"
```

Interfaces fluidos o Fluent Interfaces

Posiblemente te hayas fijado que los métodos que hemos visto aquí y la mayoría de métodos de la [documentación oficial](#) retornan un `StringBuilder`.

- ¿Para que hacer esto, si el propio objeto llamador es el que se modifica?
- ¿A caso retorna una nueva instancia de `StringBuilder` con la modificación cómo sucedía con `string`?

Lo que dice la documentación oficial de los métodos sobre el objeto `StringBuilder` de retorno es: **"Referencia a la instancia después de que se complete la operación de inserción."**

Significa que retorna la misma referencia al objeto al que aplicamos el método. Esto es, si hacemos `cadenaMutable.Replace("Juanjo", "Xusa");` esta llamada retornará el propio `cadenaMutable`.

En ocasiones es común encontrar este **patrón** en ciertos API de algunas clases y se denomina **Fluent Interfaces**. Pero, ¿Para qué se hace?

La idea es poder encadenar llamadas a métodos de modificación del objeto sin tener que repetir el identificador del mismo.

Ejemplo 1:

Imaginemos que la siguiente cadena: *"Texto a modificar"* y queremos transformarla a **html** de la siguiente forma:


```
<p>
    Texto a <b>modificar<b>
</p>
```

Una opción posible usando `StringBuilder` sería la siguiente:

```
StringBuilder htmlBuilder = new StringBuilder("Texto a modificar");
htmlBuilder.Insert(0, "<p>\n\t");
htmlBuilder.Replace("modificar", "<b>modificar<b>");
htmlBuilder.Append("\n</p>\n");
string html = htmlBuilder.ToString();
Console.WriteLine(html);
```

Pero el **interfaz fluido** de `StringBuilder` también nos permitirá hacer la implementación encadenando llamadas de la siguiente forma:

```
string html = new StringBuilder("Texto a modificar")
    .Insert(0, "<p>\n\t")
    .Replace("modificar", "<b>modificar<b>")
    .Append("\n</p>\n")
    .ToString();
Console.WriteLine(html);
```

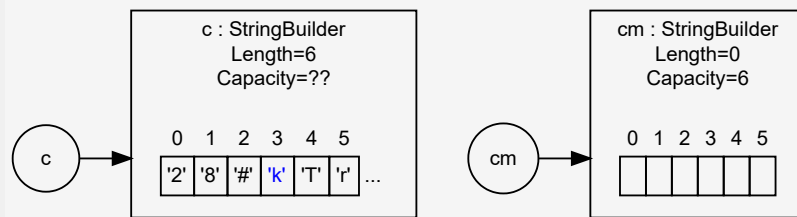
 **Tip:** Si nos colocamos con el cursor antes del operador `'.'` y vemos la opción de refactorización que nos ofrece VSCode con **Ctrl+.** . Nos ofrecerá la opción *"Encapsular cadena de Llamadas"* que alinea todas las llamadas tal y cómo se ve en el ejemplo.

Ejemplo 2:

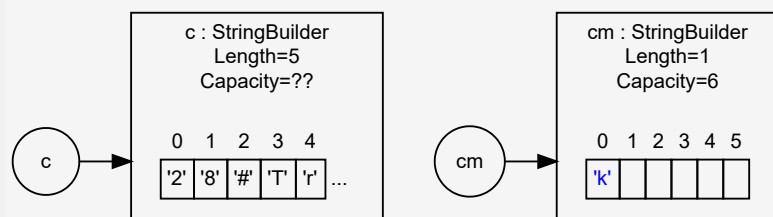
- Crea una aplicación generadora de contraseñas.
- Para ello, debes pedir cuantos **dígitos** de 0 a 9, **caracteres especiales** (`_-%/^[^><#`) y **letras** debe contener. Las letras se generarán aleatoriamente en mayúscula o en minúscula con una probabilidad del 50%.
- La longitud total de la misma será la suma de los tres valores introducidos.
- Además, solicitaremos al usuario cuantas contraseñas desea generar.
- Por último, mostraremos cada una de las contraseñas generadas.

💡 **Pista:** Generaremos los diferentes tipos de carácter siempre en el mismo orden, para simplificar el algoritmo, pero antes de devolver la contraseña generada mezclaremos los mismos ayudándonos de objetos de tipo `StringBuilder`.

Para ello supongamos que tenemos una clave con **2** dígitos, **1** carácter especial y **3** letras en un `StringBuilder` referenciado por el id `c` y creamos otro `StringBuilder` con una capacidad inicial de la longitud de nuestra clave que es **6**, referenciado por el id `cm` y en el cual iremos generando la clave mezclada.



Ahora eligiéremos aleatoriamente un carácter de `c` lo añadiremos a `cm` con `cm.Append(...)` y lo eliminaremos con `c.Remove(...)`. Supongamos que elegimos aleatoriamente el índice **3** donde se encuentra el carácter **'k'**.



Este proceso lo podremos repetir hasta que la `c.Length` sea 0

```
static class Program
{
    static char GeneraLetra(Random seed)
    {
        const string LETRAS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        char letra = LETRAS[seed.Next(0, LETRAS.Length)];
        return seed.Next(2) == 0 ? letra : char.ToLower(letra);
    }

    static char GeneraDigito(Random seed)
    {
        const string DIGITOS = "0123456789";
        return DIGITOS[seed.Next(0, DIGITOS.Length)];
    }

    static char GeneraEspecial(Random seed)
    {
        const string ESPECIALES = "_-%/^[^><#";
        return ESPECIALES[seed.Next(0, ESPECIALES.Length)];
    }
}
```

```

static string Mezcla(Random seed, string texto)
{
    StringBuilder c = new StringBuilder(texto);
    StringBuilder cm = new StringBuilder(c.Length);
    while (c.Length > 0)
    {
        int i = seed.Next(0, c.Length);
        cm.Append(c[i]);
        c.Remove(i, 1);
    }
    return cm.ToString();
}

static string ContraseñaAleatoriaParametrizada(Random seed, int dígitos, int especiales, int letras)
{
    int longitud = dígitos + especiales + letras;
    StringBuilder contraseña = new StringBuilder(longitud);
    int cDígitos = 0;
    int cEspeciales = 0;
    for (int i = 0; i < longitud; i++)
    {
        char? c;

        if (cDígitos < dígitos)
        {
            c = GeneraDigito(seed);
            cDígitos++;
        }
        else if (cEspeciales < especiales)
        {
            c = GeneraEspecial(seed);
            cEspeciales++;
        }
        else
        {
            c = GeneraLetra(seed);
            contraseña.Append(c);
        }
    }

    return Mezcla(seed, contraseña.ToString());
}

public static void Main()
{
    Console.WriteLine("Parametriza la generación de claves...");
    Console.Write("¿Cuántos dígitos contiene?: ");
    int dígitos = int.Parse(Console.ReadLine());
    Console.Write("¿Cuántos caracteres especiales?: ");
    int especiales = int.Parse(Console.ReadLine());
    Console.Write("¿Cuántas letras?: ");
    int letras = int.Parse(Console.ReadLine());
    Console.Write("¿Cuántas claves quieres generar?: ");
    int claves = int.Parse(Console.ReadLine());

    // Generamos la semilla aleatoria aquí para no obtener siempre las mismas contraseñas.
    // Esta semilla, la iremos pasando a través de los diferentes módulos que generan
    Random seed = new Random();

    for (int i = 0; i < claves; i++)
        Console.WriteLine(ContraseñaAleatoriaParametrizada(seed, dígitos, especiales, letras));
}
}

```

Caso de estudio


A continuación tienes un programa que irá pidiendo frases hasta que se introduzca la frase "fin" .

Cada una de las frases las pasará por el método estático `string TraduceALTiko(string frase)` que me devolverá la frase modificada con una jerga de mensajería instantánea que usa la ficticia '*tribu urbana*' de los **tikos**.

Analiza el código he intenta deducir las reglas que se han usado para la traducción.

Puedes probar las frases:

- ¿Has probado el Chimichurri?
- Mi socio dice que está muy bueno por Elche

 Fíjate como se ha autodocumentado el código a través de variables locales, constantes y funciones.

```
static class Program
{
    static void PasaLetrasAMayusculasAleatoriamente(StringBuilder frase, int pocenetaje)
    {
        Random seed = new Random();
        for (int i = 0; i < frase.Length; i++)
        {
            if (seed.NextDouble() < pocenetaje / 100d)
                frase[i] = char.ToUpper(frase[i]);
        }
    }

    static string TraduceALTiko(string frase)
    {
        frase = frase.ToLower();
        StringBuilder fraseTiko = new StringBuilder(frase.Length * 2);
        Random seed = new Random();

        for (int i = 0; i < frase.Length; i++)
        {
            const string VOCAL = "aeiuo";
            char c = frase[i];
            string sustitución;

            bool esLetraSiguienteEoI = i + 1 < frase.Length && (frase[i + 1] == 'e' || frase[i + 1] == 'i');
            bool esLetraSiguienteAoO = i + 1 < frase.Length && (frase[i + 1] == 'a' || frase[i + 1] == 'o');
            bool esLetraAnteriorVocal = i > 0 && VOCAL.IndexOf(frase[i - 1]) >= 0;
            bool esUltimaLetra = i == frase.Length - 1;
            switch (c)
            {
                case 'h':
                case '¿':
                case 'd' when esLetraAnteriorVocal && esLetraSiguienteAoO:
                case '?' when !esUltimaLetra:
                case '!' when !esUltimaLetra:
                    sustitución = "";
                    break;
                case '?' when esUltimaLetra:
                    sustitución = $" lokooo{new string('?', seed.Next(2, 6))}";
                    break;
                case '!' when esUltimaLetra:
                    sustitución = $" ermanooo{new string('!', seed.Next(2, 6))}";
                    break;
                case 'c' when i == 0:
                    sustitución = "k";
                    break;
                case 'c' when i + 1 < frase.Length && frase[i + 1] == 'h':
                    sustitución = "x";
                    break;
                case 'c' when esLetraSiguienteEoI:
                    sustitución = "s";
                    break;
            }
            fraseTiko.Append(sustitución);
        }
        return fraseTiko.ToString();
    }
}
```

```

        default:
        {
            StringBuilder auxiliar = new StringBuilder();
            bool esUnaPalabraQueAcabaEnVocal = i < frase.Length - 1
                                                && VOCAL.IndexOf(c) >= 0
                                                && char.IsSeparator> (frase[i + 1]);

            if (!esUnaPalabraQueAcabaEnVocal)
            {
                const string LETRAS = "gbvzáéíóú";
                const string EQUIVALENTE_LETRAS_EN_TIKO = "jvbsaeiou";
                int pos = LETRAS.IndexOf(c);
                auxiliar.Append(pos >= 0 ? EQUIVALENTE_LETRAS_EN_TIKO[pos] : c);
            }
            else
            {
                auxiliar.Append(c, seed.Next(1, 5));
            }

            if (esUltimaLetra)
                auxiliar.Append(" ermanoo");

            sustitución = auxiliar.ToString();
            break;
        }
    }
    fraseTiko.Append(sustitución);
}

PasaLetrasAMayusculasAleatoriamente(
    fraseTiko.Replace(" por ", " x ")
               .Replace(" que ", " k ")
               .Replace(" muy ", " to "),
    20);

return fraseTiko.ToString();
}

public static void Main()
{
    while (true)
    {
        Console.Write("Introduce una frase a traducir (fin para acabar): ");
        string frase = Console.ReadLine();

        string fraseTiko = frase.ToUpper() != "FIN" ? TraduceALTiko(frase) : "aDió soSssio !!!";
        Console.WriteLine(fraseTiko);
    }
}
}

```

Colecciones Homogéneas de Tamaño Fijo

Tablas Unidimensionales o (Arrays/Vectores)

Organización de datos que se caracteriza porque todos los componentes:

- Son del mismo tipo (**homogénea**).
- Se pueden acceder arbitrariamente y son igualmente accesibles (**acceso directo de coste O(1)**).

Instanciar objetos array

```
// Este objeto tabla referenciaría a null porque no se ha instanciado en memoria
// Solo estamos indicando el tipo de elementos que va a contener.
<Tipo>[] <identificadorTabla>;
```

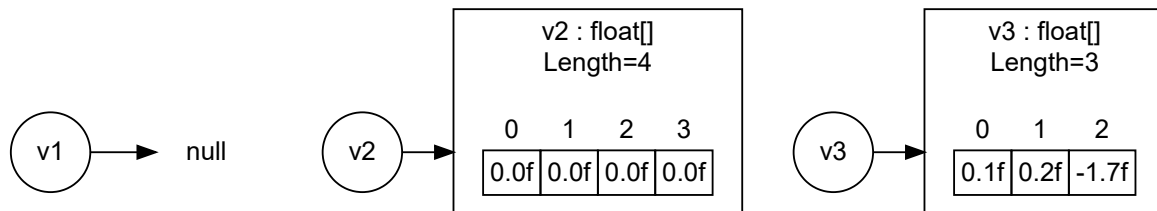
```
float[] v1;
float[] v1 = default;
```

```
// Este crearemos un objeto tabla con espacio en su interior para guardar númeroElementos del tipo definido.
// Los elementos dentro del tipo tomarán el valor default para el tipo.
<Tipo>[] <identificadorTabla> = new <Tipo>[<númeroElementos>];
```

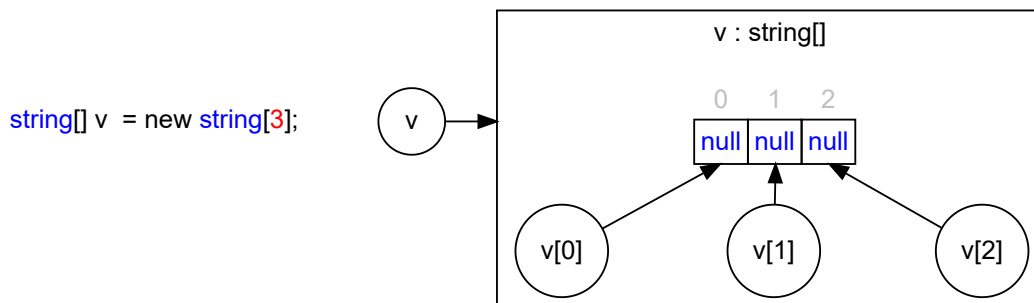
```
// Todos los elementos valen 0.0f por ser float Type-safe y su valor 'default' ser ese.
float[] v2 = new float[4];
// ERROR float[] v2 = new float[];
```

```
// En este crearemos un objeto tabla con espacio en su interior para guardar númeroElementos
// y además estamos definiendo por extensión cada elemento.
// númeroElementos será opcional y si se pone debe coincidir con el número de elementos
<Tipo>[] <identificadorTabla> = new <Tipo>[<númeroElementosOpcional>] { dato1, dato2, ... , datoN };
```

```
float[] v3 = new float[] { 0.1f, 0.2f, -1.7f };
float[] v3 = new float[3] { 0.1f, 0.2f, -1.7f }; // Posible pero no necesario.
// ERROR float[] v3 = new float[4] { 0.1f, 0.2f, -1.7f, 0.0f, 3.14f };
```



👉 **Importante:** Si el tipo de los elementos del array **es un tipo referencia** los valores dentro del mismo se oficializarán a referencias a **null**. Puesto que tras reservar el espacio, aún no se habrán instanciado cada uno de los objetos.



Indexación o acceso a los elementos de un array

Será completamente análoga a la de las cadenas. Recordemos...

- Los elementos de un "array" responden a un nombre de variable común y se identifican por el valor de una expresión entera, escrita entre corchetes (operador []), llamada índice.
- Esta expresión entera nos servirá de índice comenzando desde cero y si accedemos más allá del tamaño dimensionado se producirá el error `OutOfBoundsException`
 - Límite inferior del índice = **0**
 - Límite superior del índice = **Longitud-1**

Recorrer arrays

Será completamente análogo a la de las cadenas. Recordemos...

- Modificando el valor de un índice se puede recorrer toda la estructura.
- Podremos saber la longitud de un array dimensionado en la inicialización a través de la propiedad `v.Length`.

Ejemplo:

Recorrido de una array de doubles para sumar los valores que guarda.

```
double[] v = new double[]{2.0d, 4.0d, 5.0d, 6.0d};
double suma = 0.0d;

// Recorrido con for
6 for (int i = 0; i < v.Length; ++i)
{
    suma += v[i];
}

// Recorrido con foreach
11 foreach (double elemento in vector)
{
    suma += elemento;
}
```

Operaciones de interés con arrays

- Una de las funcionalidades que me ofrecen los arrays es la posibilidad de definir un **número indeterminado de parámetros formales** del mismo tipo en la signatura de los métodos. Lo haremos anteponiendo la palabra reservada `params` a un parámetro formal de tipo array.

```
static class Ejemplo
{
    static double Media(params double[]? valores)
    {
        double media;
        if (valores.Length > 0)
        {
            double suma = default;
            foreach (double v in valores)
                suma += v;
            media = suma / valores.Length;
        }
        else
            media = 0D;
        return media;
    }

    static void Main()
    {
        // Podré llamar de las siguiente formas:
        Console.WriteLine(Media(new double[]{2D, 5D, 7D}));
        22 Console.WriteLine(Media(2D, 5D, 7D));
    }
}
```

- Podemos ver ahora un par de métodos de utilidad en **System.string** relacionados con arrays...

- static string Join(<separador>, array)**

Me permite 'unir' los elementos de un array a través de un separador para representarlos o enumerarlos.

<separador> puede ser un **string** o un **char**

Retornará una cadena con los elementos del array separados por el separador.

- string[] Split(params char[]? separadores, opcionesDeTroceado)**

Es un método de instancia y por tanto se aplicará a un objeto cadena.

Devolverá un array con el resultado de 'trocear' o dividir el objeto cadena al que lo aplicamos, por los caracteres de separación que recibe como parámetros.

🔗 **Nota:** En **opcionesDeTroceado** podemos usar **StringSplitOptions.RemoveEmptyEntries** para evitar que tras el troceado queden cadenas vacías.

```
// Partimos con la siguiente cadena con los días de la semana separados de forma heterogénea
string t = "lunes, martes, miércoles;jueves :viernes";

string[] dias = t.Split(";;: ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
// También podríamos haber hecho
// string[] dias = t.Split(' ',';',':',',');

8 // Mostrará "lunes, martes, miércoles, jueves, viernes" homogeneizado
Console.WriteLine(string.Join(", ", dias));
```

- Dispondremos de la clase **System.Array** que implementará una **serie de métodos estáticos** de utilidad para trabajar con arrays.

Los más útiles para nosotros a estas alturas del curso serán:

- static int IndexOf(array, <elemento>)**

Análogo al IndexOf en cadenas. Busca **linealmente** $O(n)$ un elemento en el array y retorna el índice a la posición donde se encuentra o -1 si no lo encuentra.

- static void Copy(arrayOrigen, arrayDestino, int longitud)**

Copia en un objeto array **arrayDestino** ya creado y dimensionado con un tamaño igual o mayor al de **arrayOrigen**, **longitud** elementos de **arrayOrigen**.

Nota: La copia es **superficial**.

- static void Sort(array)**

Ordena el array que recibe como parámetro, pero *'solo si el contenido del array es un tipo básico como int, double, string, etc...'*

Nota: En realidad su funcionamiento es más complejo, pero aún no podemos abordarlo.

- static void Resize(ref array, int nuevoTamaño)**

Recibe una referencia a un tipo array. Si la referencia es null, crea un nuevo array con el tamaño especificado y si el objeto array ya estaba creado lo redimensiona con el tamaño especificado y devuelve una nueva referencia.

Nota: Es una **operación muy costosa** y no deberíamos usarla mucho. En los casos en que haya que redimensionar es mejor usar otras colecciones que ya estudiaremos más adelante.

- static void Clear(array, int index, int length)**

'Borra' elementos en un array. Esto es, los deja al valor **default** del tipo que contenga.

```
int[] v1 = new int[] { 3, 2, 4 };

Console.WriteLine(string.Join(", ", v1)); // Muestra 3, 2, 4
Console.WriteLine(Array.IndexOf(v1, 4)); // Muestra 2
Console.WriteLine(Array.IndexOf(v1, 5)); // Muestra -1

int[] v2 = new int[v1.Length];
Array.Copy(v1, v2, v1.Length);
Array.Sort(v2);
Array.Resize(ref v2, v2.Length + 1);
v2[v2.Length - 1] = 5;
Console.WriteLine(string.Join(", ", v2)); // Muestra 2, 3, 4, 5
Console.WriteLine(string.Join(", ", v1)); // Muestra 3, 2, 4
Array.Clear(v1, 0, v1.Length);
Console.WriteLine(string.Join(", ", v1)); // Muestra 0, 0, 0
```

👉 Pasando y devolviendo arrays en métodos

En este apartado pretendemos hacer una reflexión, sobre cómo trabajar con arrays cuando definamos la signatura de un método. Para ellos, vamos a verlo a través de un ejemplo...

Supongamos que tenemos un array de cadenas con verbos en inglés.

```
string[] verbs = new string[]
{
    "be", "eat", "see"
};
```

Ahora queremos definir un método que modifique el contenido del array para que anteponga la cláusula `"to ..."` a cada verbo devolviéndonos `["to be", "to eat", "to see"]`

Si implementamos la interfaz de la siguiente implementación...

```
class Ejemplo
{
    static void AddVerbPrefix(string[] verbs)
    {
        for (int i = 0; i < verbs.Length; i++)
            verbs[i] = $"to {verbs[i]}";

        /*
        Cuidado !!!! la siguiente implementación ...

        foreach(string verb in verbs)
            verb = $"to {verbs[i]}";

        No sería válida porque no estamos modificando el contenido del array.
        Si lo piensas, las referencias en el array sería las mismas porque no estamos
        modificándolas a través del indizador.
        */
    }

    static void Main()
    {
        string[] verbs = new string[] { "be", "eat", "see" };

        // Pasamos una copia de la referencia al objeto string[] apuntada por verbs.
        AddVerbPrefix(verbs);
        Console.WriteLine(string.Join(", ", verbs));
    }
}
```

Si nos fijamos en la salida, como `AddVerbPrefix(verbs);` no retorna nada, **solo con ver el interfaz** y sin saber cómo está implementado el método, podemos deducir que es el contenido del objetos `verbs` el que se ha modificando añadiéndose el prefijo "to" a los verbos del array que define, y por tanto **perdiendo el contenido original** donde teníamos los verbos sin prefijo.

Pero... **¿Cómo lo implementaríamos si queremos que me cree un nuevo array de verbos sin modificar el original?**


```

class Ejemplo
{
    static string[] AddVerbPrefix(string[] verbs)
    {
        // Dimensionamos el array donde irán las cadenas modificadas.
        // recuerda que los objetos cadena que contiene no están definidos
        // y apuntarán a null.
        string[] verbsWithPrefix = new string[verbs.Length];

        for (int i = 0; i < verbsWithPrefix.Length; i++)
            // Instancio la nueva cadena con prefijo en las posiciones del array.
            verbsWithPrefix[i] = $"to {verbs[i]}";

        // retorno la referencia al array. Deberá ser siempre un nuevo objeto instanciado en memoria.
        return verbsWithPrefix;
    }
    static void Main()
    {
        string[] verbs = new string[] { "be", "eat", "see" };
        string[] verbsWithPrefix = AddVerbPrefix(verbs);
        Console.WriteLine(string.Join(", ", verbs));
        Console.WriteLine(string.Join(", ", verbsWithPrefix));
    }
}

```

Si nos fijamos en la salida ambos arrays tendrán contenido diferente y de esta forma no habremos perdido el array original. Eso sí, asumiendo el coste de instanciar y crear uno nuevo.

Además, siempre que veamos una llamada en la que se retorna un objeto `string[] verbsWithPrefix = AddVerbPrefix(verbs);` deberemos deducir que es un método *'factoria'* esto es, retorna un objeto nuevo y **no el que se le pasó como referencia** lo cual puede ser peligroso. Supongamos que hacemos...

```

class Ejemplo
{
    // No confundir esto con lo que se pretendía hacer en el patrón fluent interface, pues es completamente diferente. Ya que es un mét
    static string[] AddVerbPrefix(string[] verbs)
    {
        for (int i = 0; i < verbs.Length; i++)
            verbs[i] = $"to {verbs[i]}";

        // Aquí estoy devolviendo la misma referencia que recibo como parámetro. (MALA PRÁCTICA)
        return verbs;
    }

    static void Main()
    {
        string[] verbs = new string[] { "be", "eat", "see" };
        string[] verbsWithPrefix = AddVerbPrefix(verbs);

        // Ahora estoy modificando los 2 arrays. (ALIASING)
        verbsWithPrefix[0] = "to sit";

        Console.WriteLine(string.Join(", ", verbs));
        Console.WriteLine(string.Join(", ", verbsWithPrefix));
    }
}

```

Ahora, además de perder el array con los verbos sin prefijo, `verbs` y `verbsWithPrefix` son una referencia al mismo objeto array en memoria. Por lo que si modifico el contenido de uno, también modifico el del otro. Produciéndose un efecto denominado **'aliasing'** 🤖.

Nota: Recuerda que el que usa mi método no tiene por qué conocer su implementación y posiblemente sumirá que `string[] AddVerbPrefix(string[] verbs)` me devuelve un array nuevo.

Caso de estudio

Aunque ya hemos visto que el lenguaje ya implementa un método de utilidad para ordenación de arrays como es `Array.Sort()`. Vamos a ver un ejemplo de recorrido e intercambio de elementos en un array a través de un ejemplo y estudiando uno de los algoritmos básicos de ordenación de arrays, como es el de la '**burbuja**' ([bubble sort](#)).

En este algoritmo, recorreremos el array **comparando 2 a 2 los elementos contiguos del mismo**. De forma que intercambiaremos cuando un elemento sea mayor que su sucesor, así en un recorrido el elemento mayor promocionará hasta el final del array, por esto se denomina de burbuja porque se dice que '*asciende*' dentro del array como si lo fuera.

Una vez ha ascendido un elemento este queda **fijo**, y volveremos a comparar 2 a 2 los elementos sin tomar el último, de tal manera que ahora ascenderá o promocionará el elemento anterior.

Este proceso se repetirá sucesivamente, teniendo en cuenta que **no tenemos que comparar con los ya promocionados o fijados**.

```
static class Ejemplo
{
    static int[] Ordena(int[] array)
    {
        int[] arrayOrdenado = new int[array.Length];
        Array.Copy(array, arrayOrdenado, array.Length);

        for (int i = 1; i < arrayOrdenado.Length; i++)
        {
            for (int j = 0; j < arrayOrdenado.Length - 1 - i; j++)
            {
                if (arrayOrdenado[j] > arrayOrdenado[j + 1])
                {
                    // Proceso de intercambio o swap de dos 'celdas' contiguas del array
                    int auxiliar = arrayOrdenado[j];
                    arrayOrdenado[j] = arrayOrdenado[j + 1];
                    arrayOrdenado[j + 1] = auxiliar;
                }
            }
        }
        return arrayOrdenado;
    }
    static void Main()
    {
        int[] array = new int[] { 5, 6, 4, 2, 3, 1 };
        int[] arrayOrdenado = Ordena(array);
        Console.WriteLine($"Array original: {string.Join(", ", array)}");
        Console.WriteLine($"Array ordenado: {string.Join(", ", arrayOrdenado)}");
    }
}
```

Recorrido 1 (i = 1)-----

[5, 6, 4, 2, 3, 1]

[5, 6, 4, 2, 3, 1] → [5, 4, 6, 2, 3, 1] Intercambio

[5, 6, 6, 2, 3, 1] → [5, 4, 2, 6, 3, 1] Intercambio

[5, 4, 2, 6, 3, 1] → [5, 4, 2, 3, 6, 1] Intercambio

[5, 4, 2, 3, 6, 1] → [5, 4, 2, 3, 1, 6] Intercambio

Recorrido 2 (i = 2)-----

[5, 4, 2, 3, 1, 6] → [4, 5, 2, 3, 1, 6] Intercambio

[4, 5, 2, 3, 1, 6] → [4, 2, 5, 3, 1, 6] Intercambio

[4, 2, 5, 3, 1, 6] → [4, 2, 3, 5, 1, 6] Intercambio

[4, 2, 3, 5, 1, 6] → [4, 2, 3, 1, 5, 6] Intercambio

Recorrido 3 (i = 3)-----

[4, 2, 3, 1, 5, 6] → [2, 4, 3, 1, 5, 6] Intercambio

[2, 4, 3, 1, 5, 6] → [2, 3, 4, 1, 5, 6] Intercambio

[2, 3, 4, 1, 5, 6] → [2, 3, 1, 4, 5, 6] Intercambio

Recorrido 4 (i = 4)-----

[2, 3, 1, 4, 5, 6]

[2, 3, 1, 4, 5, 6] → [2, 1, 3, 4, 5, 6] Intercambio

Recorrido 5 (i = 5)-----

[2, 1, 3, 4, 5, 6] → [1, 2, 3, 4, 5, 6] Intercambio

[1, 2, 3, 4, 5, 6]

Tablas Multidimensionales

Matrices

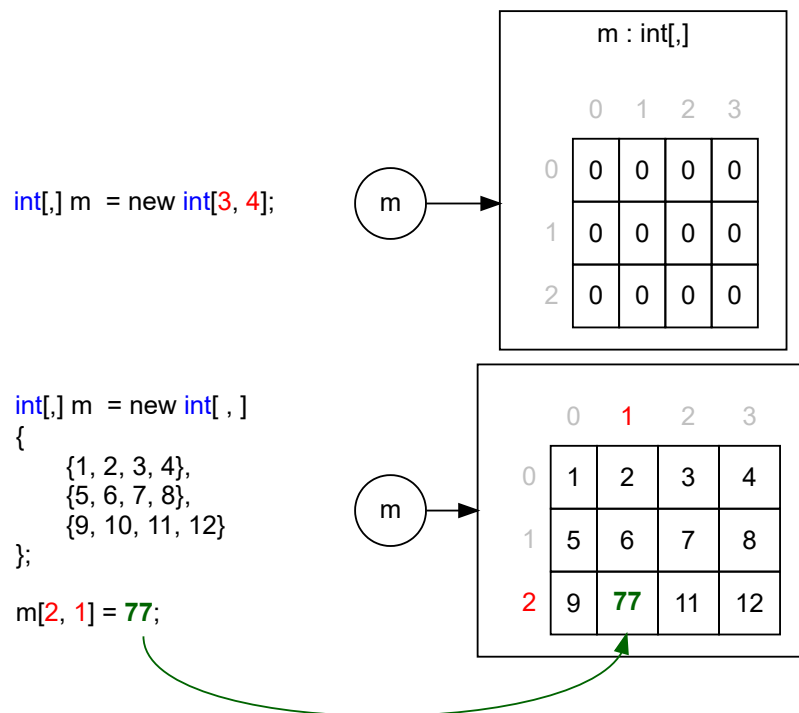
- A las colecciones de 2 dimensiones las denominaremos **matrices**.
- ☞ Intentaremos **evitar** soluciones con **tipos de datos de más de 2 dimensiones**.
Pues suelen dar lugar a código ofuscado difícil de mantener 🤖.

Instanciar objetos de tipo matriz

- Las dimensiones se añaden de **derecha a izquierda** separadas por comas.
[..., z, y, x] → [..., matriz/profundidad, fila, columna]
- Podremos definir las de varias maneras, inicializando a los valores del usuario.
- Pero ara mayor claridad, es importante que la definición se trate de separar en varias líneas tabuladas.

```
int[,] matriz = 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[,] matriz =
{
    {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.WriteLine($"{matriz[i, j],-4}");
        }
        Console.WriteLine("\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**.

Caso de estudio

El código siguiente define una matriz tri-dimensional de enteros no repetidos y selecciona en la misma un valor al azar. A continuación, le preguntará al usuario los valores de los índices para acceder al mismo. Si falla le indicará el fallo y los valores correctos que debería haber introducido.

El ejemplo es complejo y propone una representación de la '*profundidad*' o tercera dimensión dibujando a las derecha las matrices de mayor índice en la tabla.

Nota: Fíjate en los recorridos, la forma de acceder a un elemento, la forma de saber la longitud de cada dimensión, índices usados, la modularización e interfaces propuestos, etc.

```
using System;

namespace Ejemplo
{
    static class Ejemplo
    {
        static void DibujaPosicion(
            in int valor,
            in int i, in int j, in int k,
            in int columnasPorMatriz, in int filasPorMatriz,
            ConsoleColor color)
        {
            Console.ForegroundColor = color;
            const int MAXIMO_DIGITOS_ENTERO = 3;
            int offsetColumna = k * MAXIMO_DIGITOS_ENTERO;
            int offsetMatriz = columnasPorMatriz * (MAXIMO_DIGITOS_ENTERO + 1) * i;
            Console.SetCursorPosition(offsetColumna + offsetMatriz, j);
            Console.Write($"{valor} , MAXIMO_DIGITOS_ENTERO:D}");
            Console.ForegroundColor = ConsoleColor.White;
            Console.SetCursorPosition(0, filasPorMatriz + 2);
        }

        static void Dibuja(int[, ,] tabla)
        {
            Console.Clear();
            for (int i = 0; i < tabla.GetLength(0); i++) // matriz
                for (int j = 0; j < tabla.GetLength(1); j++) // fila
                    for (int k = 0; k < tabla.GetLength(2); k++) // columna
                        {
                            DibujaPosicion(
                                tabla[i, j, k], i, j, k,
                                tabla.GetLength(2), tabla.GetLength(1),
                                ConsoleColor.White);
                        }
        }

        static void MuestraResultado(
            int[, ,] tabla, in int i, in int j, in int k,
            in int iUsuario, in int jUsuario, in int kUsuario)
        {
            string mensaje;
            Dibuja(tabla);
            bool concidenPosiciones = i == iUsuario
                && j == jUsuario
                && k == kUsuario;
            DibujaPosicion(
                tabla[i, j, k],
                i, j, k, tabla.GetLength(2), tabla.GetLength(1),
                ConsoleColor.Green);
            if (!concidenPosiciones)
                DibujaPosicion(
                    tabla[iUsuario, jUsuario, kUsuario],
                    iUsuario, jUsuario, kUsuario, tabla.GetLength(2), tabla.GetLength(1),
                    ConsoleColor.Red);
            mensaje = concidenPosiciones ? "Enhorabuena has acertado!!" : $"Lo siento los indices correctos son [{i}, {j}, {k}]";
            Console.WriteLine(mensaje);
        }
    }
}
```

```

static (int p, int f, int c) PosicionAleatoria(
    in int matricesTotales,
    in int filasTotales,
    in int columnasTotales)
{
    Random s = new Random();
    return (s.Next(0, matricesTotales), s.Next(0, filasTotales), s.Next(0, columnasTotales));
}

static int PideIndice(string nombreIndice, in int valorMaximo)
{
    int i;
    bool entradaCorrecta;
    do
    {
        Console.Write($"{nombreIndice}: ");
        entradaCorrecta = int.TryParse(Console.ReadLine(), out i);
        if (entradaCorrecta)
            entradaCorrecta = i >= 0 && i <= valorMaximo;
    } while (!entradaCorrecta);

    return i;
}

static (int i, int j, int k) PosicionUsuario(int[, ] tabla)
{
    int i = PideIndice("Índice matriz", tabla.GetLength(0) - 1);
    int j = PideIndice("Índice fila", tabla.GetLength(1) - 1);
    int k = PideIndice("Índice columna", tabla.GetLength(2) - 1);
    return (i, j, k);
}

static void Main()
{
    int[, ] tabla = new int[, ]
    {
        {
            {1,2,3,4},
            {5,6,7,8},
            {9,10,11,12}
        },
        {
            {13,14,15,16},
            {17,18,19,20},
            {21,22,23,24}
        }
    };

    do
    {
        Dibuja(tabla);
        (int i, int j, int k) = PosicionAleatoria(tabla.GetLength(0), tabla.GetLength(1), tabla.GetLength(2));

        Console.WriteLine($"Introduce los índices [iMatriz, iFila, iColumna] para acceder al valor {tabla[i, j, k]}...");
        (int iUsuario, int jUsuario, int kUsuario) = PosicionUsuario(tabla);

        MuestraResultado(
            tabla,
            i, j, k,
            iUsuario, jUsuario, kUsuario);

        Console.WriteLine("\nPulsa una tecla para otro intento o ESC para salir.");
    } while (Console.ReadKey().Key != ConsoleKey.Escape);
}
}

```

Combinando colecciones homogéneas

- Podemos hacer que el contenido de una colección homogénea sea otra colección homogénea.
- Debemos 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)

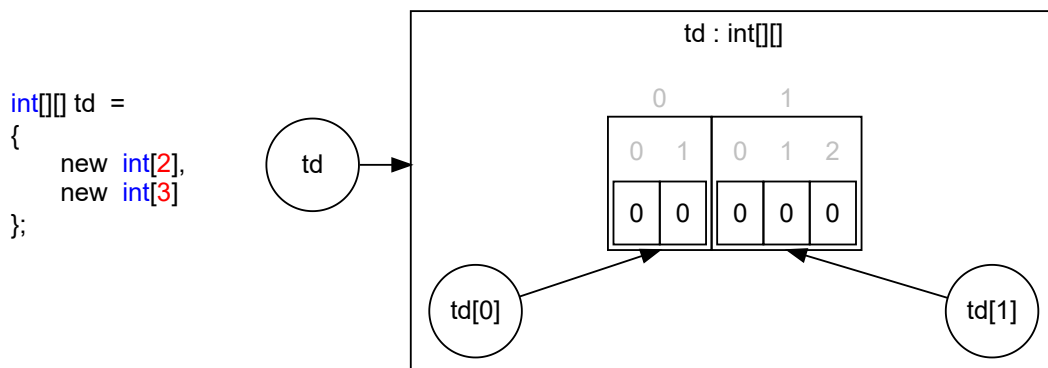
- Es una tabla de tablas o **array de arrays**
- Son estructuras utilizadas cuando necesitamos una matriz, donde la longitud de las filas no va a ser la misma. (De ahí lo de '*dentadas*').

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.

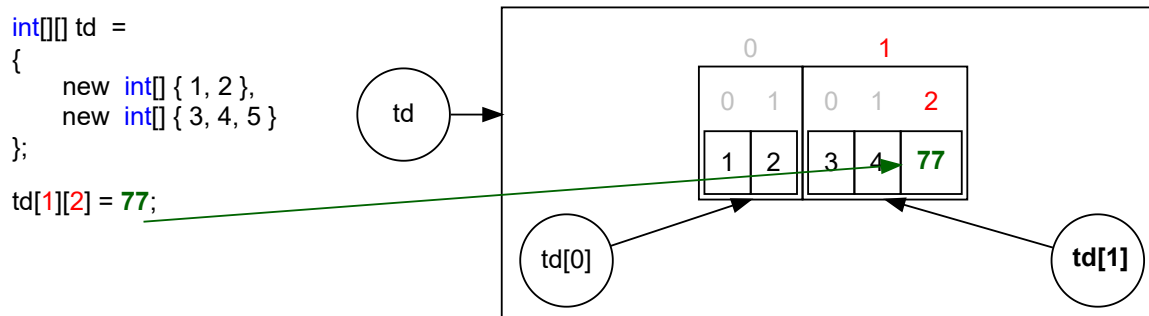
```
int[][] td =
{
    new int[] {1, 2}, // [1][2]
    new int[] {3, 4, 5}, // [3][4][5]
};
```

- Para acceder a uno de los datos, primero accederemos a la fila indizando el objeto array que lo contiene. Por ejemplo, si quisiéramos cambiar el valor **5** por un **77** accederíamos al array que contiene el 5 a través de **td[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;

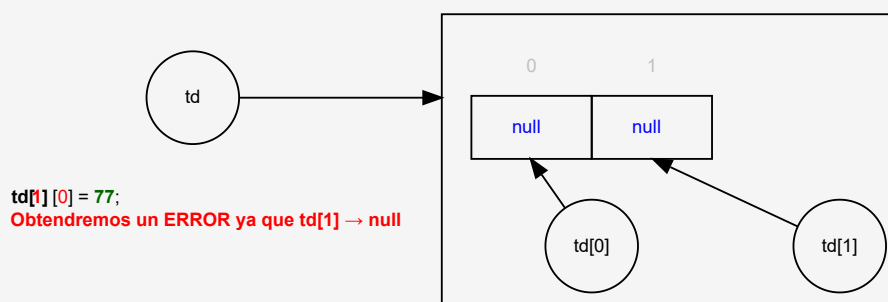
// Sería equivalente ha hacer...

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

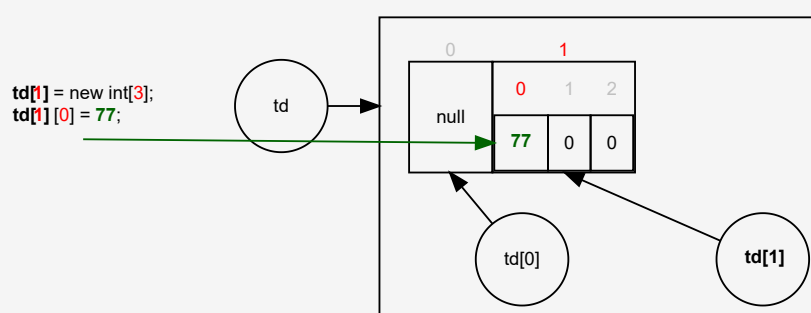


¿Qué pasa si no instanciamos o no definimos por extensión todos o alguno de los arrays de la tabla dentada?

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 instanciamos 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
11  for (int i = 0; i < td.Length; i++)
    {
13      for (int j = 0; j < td[i].Length; j++)
        Console.Write($"{td[i][j],-4}");
        Console.Write("\n\n");
    }

    // Recorrido con un doble foreach
19  foreach (int[] fila in td)
    {
21      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...

```
[Comunidad 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

```
Comunidad 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[]
        {
            "Comunidad 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: **NombreDeEnum.Identificador**
- **Ejemplos:**

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

Tamaño tamaño = default; // Equivale ha 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 a cadena un enum.
- `Enum.Parse(string id, bool ignoraMayúsculas)`
Pasa a enum una cadena.
- `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 = Enum.Parse("lunes", 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(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** que contenga un enumerado ...

```
1 switch (d)
  {
    default:
4  }
```

y pulsamos 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 booleano.

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 través 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 ver almacenar los estados combinados de una juego de plataformas. De tal manera que **la primera letra** del 'flag' me va a 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 determinado estado.

Nota: Fíjate como el código 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($"PlayerState = {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');
    }
}
```