

Tema 5.1

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

Índice

- [Índice](#)
- ▼ [Programación Orientada a Objetos Básica \(POO\)](#)
 - [Introducción](#)
 - [Definición de Tipo Abstracto De Datos o TAD](#)
 - ▼ [Definición de clase](#)
 - [Elementos que definen una clase](#)
 - ▼ [Definición de objeto](#)
 - [Composición de un objeto](#)
 - [Definición de Atributo](#)
 - [Instanciando objetos de una clase](#)
- ▼ [Estructuras de datos básicas](#)
 - ▼ [Cadenas Inmutables](#)
 - [Introducción](#)
 - [Formas de instanciar objetos cadena](#)
 - [Comparación de cadenas](#)
 - [Obtener la longitud de una cadena](#)
 - [Accediendo a los caracteres de una cadena](#)
 - [Recorrer los caracteres de una cadena](#)
 - [Operaciones de interés sobre objetos cadena](#)
 - ▼ [Cadenas Mutables](#)
 - [Instanciar un StringBuilder](#)
 - [Operaciones de interés sobre objetos cadena mutable](#)
 - [Interfaces fluidos o Fluent Interfaces](#)
 - ▼ [Colecciones Homogéneas de Tamaño Fijo](#)
 - ▼ [Tablas Unidimensionales o \(Arrays/Vectores\)](#)
 - [Instanciar objetos array](#)
 - [Indexación o acceso a los elementos de un array](#)
 - [Recorrer arrays](#)
 - [Operaciones de interés con arrays](#)
 - [Pasando y devolviendo arrays en métodos](#)
 - ▼ [Los tipos Index y Range](#)
 - [El tipo Index](#)
 - [El tipo Range](#)
 - [Operadores de indexación intervalo .. y final ^](#)
- [👉 Operadores condicionales Null ?. y ?\[\]](#)
- [Operador supersión de NULL .! \('null-forgiving'\)](#)
- [Inferencia de tipos al instanciar un objeto](#)

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.
- 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.
- Podemos considerar el paradigma Orientado a Objetos como un 'superconjunto' de la programación imperativa y estructurada. Esto es, nos proporciona más herramientas para abordar la resolución de los problemas.

Programación
Orientada a Objetos

Programación
Imperativa/Estructurada

Definición de Tipo Abstracto De Datos o TAD

- Nos preguntaremos que tipos de entidades 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.

🔴 **Nota:** Las que hemos estado creando hasta ahora y que solo contenían métodos estáticos. Un ejemplo puede ser la clase `Math` de las BCL y que solo contiene funciones de utilidad matemática. A esta organización también se las conoce como `<<Utility>>`

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
Saldo = 30000 Titular = "Xusa"

cuenta2 : Cuenta
Saldo = 30000 Titular = "Xusa"

cuenta3 : Cuenta
Saldo = 15000 Titular = "Juanjo"

cuenta1≡cuenta1
cuenta1=cuenta2
cuenta1≠cuenta3

 **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?**
 - De Instancia**
 - Serán diferentes en cada objeto.
 - Necesitaré de un objeto instanciado en memoria para acceder a ellos.
 - 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 , y devuelve una referencia a la dirección de memoria dinámica donde se ha creado el objeto.

```
Cuenta cuenta1 = new Cuenta();  
Cuenta cuenta1 = new ();           // En C# 10
```

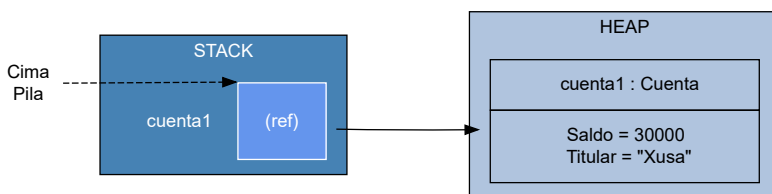
- Para acceder a los atributos y a los métodos de un objeto usaremos el operador .

```
<objeto>.<campo>  
<objeto>.<método>(<parámetros>)
```

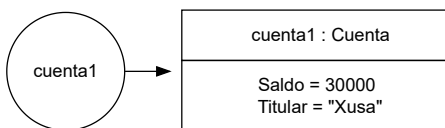
```
cuenta1.Saldo = 30000;  
cuenta1.Titular = "Xusa";
```

- Los objetos definidos por nosotros suelen ser **tipos referencia**, significa que el identificador `cuenta1` **será una referencia** en el **Stack** a una instancia en memoria de un objeto de tipo `Cuenta` en el **Heap**

🔴 **Nota:** Repasa los conceptos de **Stack** (Pila) y **Heap** (Memoria Montón) descritos al final del **Tema 2.1**

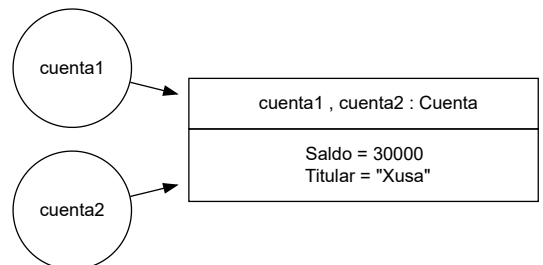


👉 **Importante:** Para simplificar, este tipo de referencias las vamos a representar a partir de ahora de la siguiente forma, que equivaldrá al esquema que hemos dibujado arriba.



- De tal manera que si hacemos ...

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

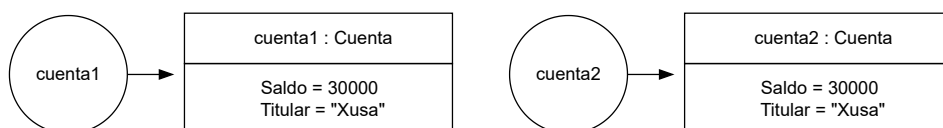


Recuerda que ambas referencias *'apuntarán'* al mismo objeto en la memoria montón.


- 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 el 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' });

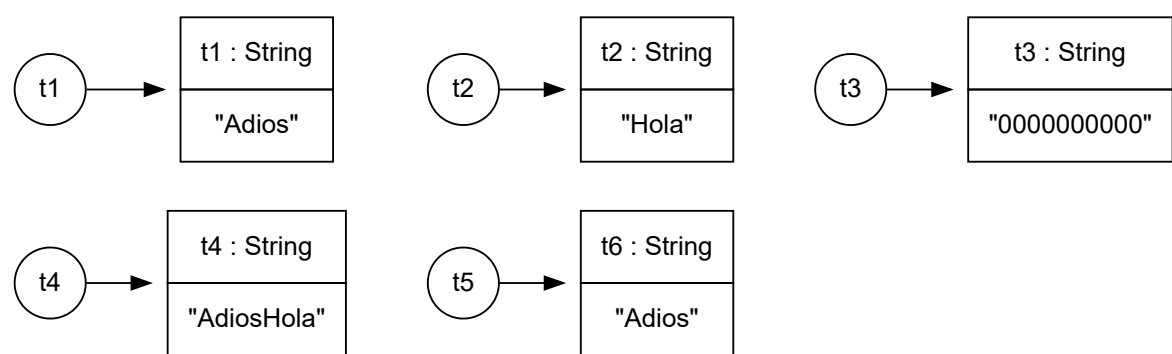
// Inicializando con un carácter de relleno.
string t3 = new String('0', 10);

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

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

// 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
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.

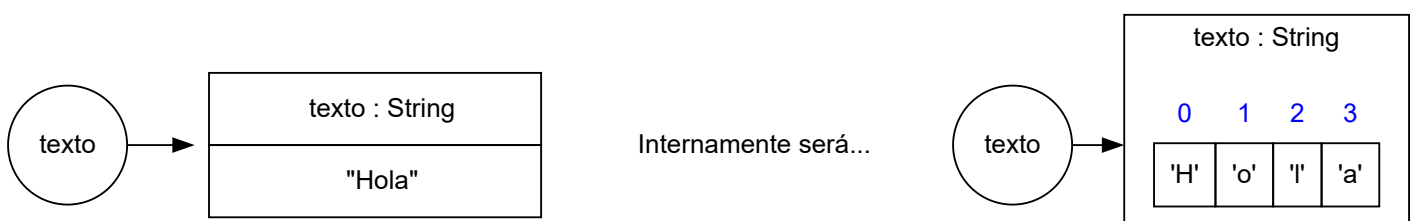
```
Console.WriteLine("Hola".Length); // Mostrará 4
string t = "Adios";
Console.WriteLine(t.Length); // Mostrará 5
```

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`.

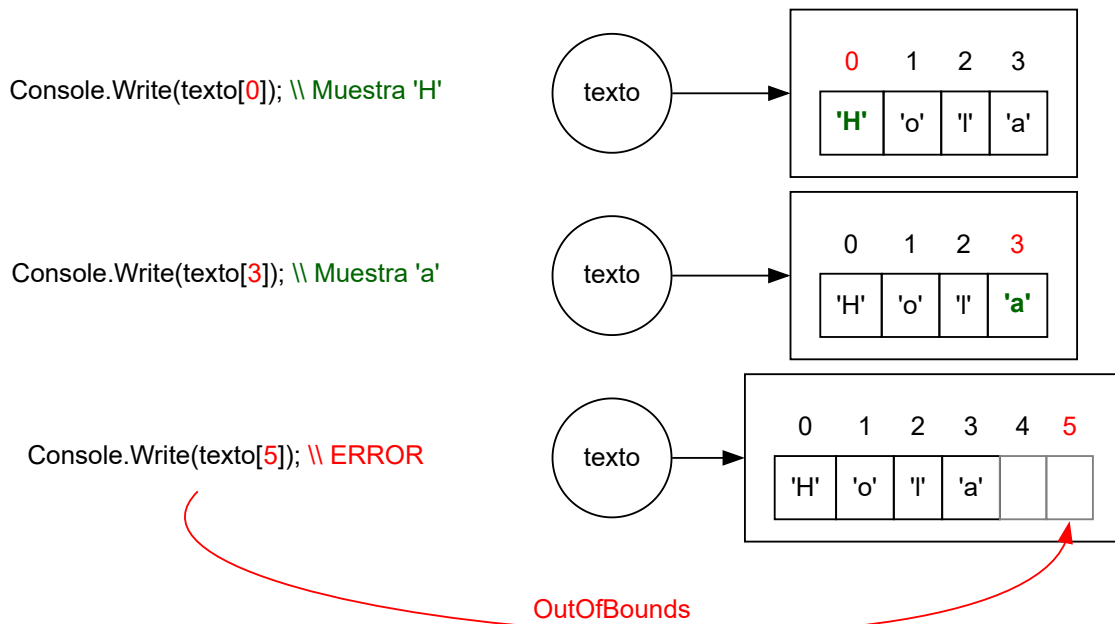
```

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

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

// 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
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 **secuencia de datos**.

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

🔴 **Nota:** En una cadena la secuencia de datos está definida por cada uno de los caracteres que la forman.


```

// Recorrido con foreach
// El bucle foreach es más apropiado en caso de no necesitar
// saber en que posición está el caracter.
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"`

```
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;
}

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().ToLower()
        };
    }
    return sN;
}
```

- **cadena.ToArray()**

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.

🔴 **Nota: Más adelante** veremos el tipo array y la utilidad de esta conversión.

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

- **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)
    {
        sN += c >= 0 ? SUSTITUCIONES[TILDES.IndexOf(c)] : 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 o tiene menos de 8 caracteres.
- **Débil:** Contiene solo letras y tiene al menos 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 2 caracteres desde el índice 3.  
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.
            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.
            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.ToLower());
        for (int i = 0; i < s.Length; i++)
        {
            int iVocal = TILDES.IndexOf(sb[i]);
            sb[i] = iVocal >= 0 ? SUSTITUCIONES[iVocal] : sb[i];
        }
        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 previamente reservada.
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 en el índice 2
cadenaMutable.Insert(2, cadena);
Console.WriteLine(cadenaMutable); // Muestra: aabbbbbaa

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

- **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 caracter 'a' por 'e'
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"
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?.

Término acuñado por Martin Fowler. 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);
```

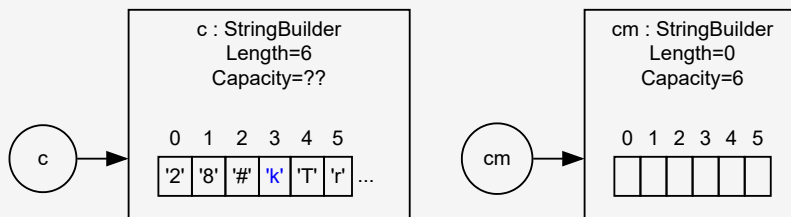
💡 **Tip:** Si nos colocamos con el cursor antes del operador `'.'` y vemos la opción de refactorización que nos ofrece VSCode con **Ctrl+.** Nos ofrecerá la opción *"Encapsular cadena de Llamadas"* que alinea todas las llamadas tal y cómo se ve en el ejemplo.

Ejemplo 2:

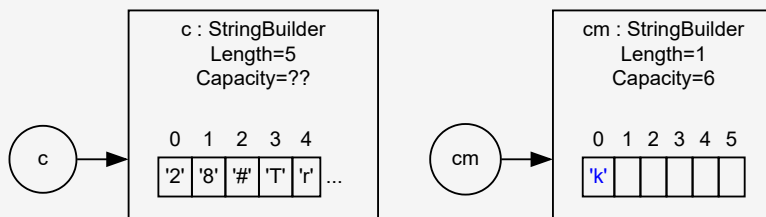
- Crea una aplicación generadora de contraseñas.
- Para ello, debes pedir cuantos **dígitos** de 0 a 9, **caracteres especiales** (`_-%/[]^><#`) y **letras** debe contener. Las letras se generarán aleatoriamente en mayúscula o en minúscula con una probabilidad del 50%.
- La longitud total de la misma 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() ?? "6");
    Console.Write("¿Cuántos caracteres especiales?: ");
    int especiales = int.Parse(Console.ReadLine() ?? "2");
    Console.Write("¿Cuántas letras?: ");
    int letras = int.Parse(Console.ReadLine() ?? "2");
    Console.Write("¿Cuántas claves quieres generar?: ");
    int claves = int.Parse(Console.ReadLine() ?? "1");

    // 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 && esLetraSiguienteAo0:
    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;

        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);
}

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

return fraseTiko.ToString();
}

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

        bool acabar = frase.ToUpper() == "FIN";
        string fraseTiko = acabar ? "aDió so$ssio !!!" : TraduceALTiko(frase);
        Console.WriteLine(fraseTiko);
        if (acabar) break;
    }
}

```

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>];

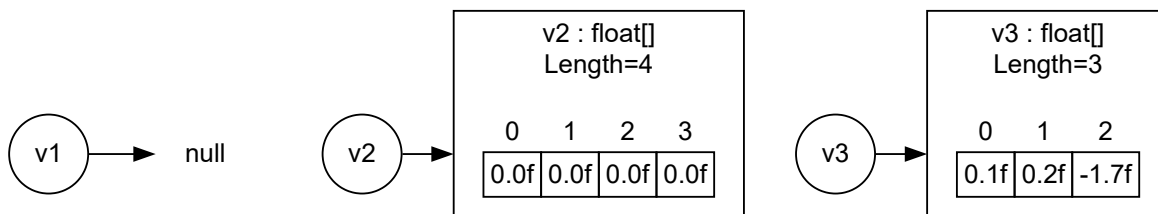
// Todos los elementos valen 0.0f por ser float Type-safe y su valor 'default' ser ese.
float[] v2 = new float[4];
// ERROR float[] v2 = new float[];
```

```
// 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 };

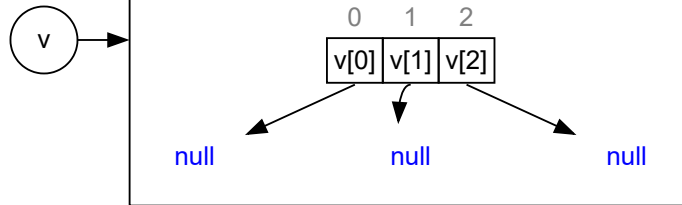
// Las versiones modernas de C# permiten este tipo de definiciones por extensión.
// Por tanto, sería LA MEJOR FORMA DE INICIALIZAR y similar a otros lenguajes.
float[] v3 = { 0.1f, 0.2f, -1.7f };

float[] v3 = new float[3] { 0.1f, 0.2f, -1.7f }; // Posible pero no necesario ni recomendables
// 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.

```
string[] v = new string[3];
```



Novedades en la inicialización de arrays en C# 11

Desde .NET 8 se han añadido al lenguaje las denominadas **expresiones de colección** estas permiten definir expresiones que se evalúan a una colección de forma similar a como se hace en otros lenguajes como **JavaScript** utilizando corchetes `[]` lo cual nos puede ser más familiar.

```
int[] edades = [20, 41, 6, 18, 23];
```

Así mismo, se pueden combinar arrays de forma sencilla utilizando el operador `..` **elemento de propagación**.

Similar al `...` (spread operator) de **JavaScript** siendo otro '*guiño*' al lenguaje.

```
int[] fila1 = [1, 2, 3];
int[] fila2 = [4, 5, 6];
int[] fila3 = [7, 8, 9];
int[] filaUnica = [.. fila1, .. fila2, .. fila3];
foreach (var valor in filaUnica)
{
    Console.WriteLine($"{valor} ");
}
// Mostará: 1 2 3 4 5 6 7 8 9
```


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 = [ 2.0d, 4.0d, 5.0d ,6.0d ];
double suma = 0.0d;
// Recorrido con for
for (int i = 0; i < v.Length; ++i)
{
    suma += v[i];
}

// Recorrido con foreach
foreach (double elemento in v)
{
    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}));
        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(',', ';', ':', ' ');

// 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` .

⚠ **Aviso:** 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...'*

⚠ **Aviso:** 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.

⚠ **Aviso:** 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 una array. Esto es, los deja al valor `default` del tipo que contenga.

```
int[] v1 = [ 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 = [ "be", "eat", "see" ];
```

Ahora queremos definir un método que modifique el contenido del array para que anteponga la cláusula `"to ..."` a cada verbo devolviéndome `["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 = [ "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?**

```

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;
}

string[] verbs = [ "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 *'factoría'* esto es, retorna un objeto nuevo y **no el que se le pasó como referencia** lo cual puede ser peligroso. Supongamos que hacemos...

```

// No confundir esto con lo que se pretendía hacer en el patrón fluent interface, pues és completamente diferente.
// Ya que es un método estático y no de instancia.
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;
}

string[] verbs = [ "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 asumirá 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 int[] Ordena(int[] array)
{
    int[] arrayOrdenado = new int[array.Length];
    Array.Copy(array, arrayOrdenado, array.Length);
    for (int i = 0; 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;
}

int[] array = [ 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 = 0)-----

[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 = 1)-----

[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 = 2)-----

[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 = 3)-----

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

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

Recorrido 5 (i = 4)-----

```
[2, 1, 3, 4, 5, 6] -> [1, 2, 3, 4, 5, 6] Intercambio  
[1, 2, 3, 4, 5, 6]
```

Los tipos Index y Range

El tipo Index

Además de con enteros, vamos a poder indexar arrays con un tipo denominado **Index** que me permitirá definir una posición en un array empezando desde el comienzo o desde el final.

```
string[] palabras = [ "cero", "uno", "dos", "tres" ];  
  
// Usamos public Index (int value, bool fromEnd = false);  
// Indice que toma el primer elemento desde el final.  
Index i = new Index(1, true);  
Console.WriteLine(palabras[i]); // Mostrará "tres"  
  
// Indice que toma el primer elemento desde el principio.  
i = new Index(0);  
Console.WriteLine(palabras[i]); // Mostrará "cero"
```

El tipo Range

Además del tipo Index, existe un tipo **Range** que podremos traducir como **rango** o **intervalo**. Que me servirá para describir un subconjunto de elementos contiguos dentro de un array desde un determinado índice de comienzo hasta uno de fin.

Son bastante útiles si queremos extraer un subconjunto de elementos de un array a partir de un par de índices que hagan de límites.

```
string[] palabras = [ "cero", "uno", "dos", "tres" ];  
  
// Cogemos el segundo elemento como inicio del intervalo y el último de forma excluyente.  
Range intervalo = new Range(new Index(1), new Index(1, true));  
string[] palabrasSinLosExtremos = palabras[intervalo];  
  
// Mostrará "uno", "dos"  
Console.WriteLine(string.Join(", ", palabrasSinLosExtremos));
```

Operadores de indexación intervalo .. y final ^

Ya conocemos los tipos **Index** y **Range** pero nosotros **no los vamos a usar así**, ya que esta forma de usarlos es poco útil y nos va a generar más código. Para ellos, existen unos operadores que nos facilitarán la labor y simplifican la sintaxis aproximándome a la de otros lenguajes modernos.

🔴 **Nota:** Hemos explicado los tipos Index y Range para que sepamos que tipos hay involucrados, pero nosotros los vamos a usar de una forma más simplificada **sin tener que hacer** `new Index()` o `new Range()` .

Indexación desde el operador final ^

Se evalúa directamente a un tipo `Index` y equivale a indicar que un índice entero se aplicará empezando desde el final del array. Veámoslo a través los siguientes ejemplos comentados...

Ejemplo 1:

```
string[] palabras =
[
    // índice desde comienzo    índice desde el final
    "cero",    // 0              ^4
    "uno",     // 1              ^3
    "dos",     // 2              ^2
    "tres",    // 3              ^1
];            // 4 (or palabras.Length) ^0

// Donde ...
Index u = new Index(1, true);
// equivale a ...
Index u = ^1;

// y por tanto las siguientes expresiones serán equivalente para
// obtener el último elemento de un array.

string p = palabras[palabras.Length - 1];
string p = palabras[new Index(1, true)];
string p = palabras[^1];
```

Ejemplo 2:

```
string[] diasSemana =
[
    //Índice desde el inicio    índice desde el final
    "Lunes",    // 0              ^7
    "Martes",   // 1              ^6
    "Miércoles", // 2              ^5
    "Jueves",   // 3              ^4
    "Viernes",  // 4              ^3
    "Sábado",   // 5              ^2
    "Domingo"   // 6              ^1
];            // 7 diasSemana.Length ^0
```

- El índice `0` representa el primer elemento
- El índice `^0` es lo mismo que `diasSemana[diasSemana.Length]`, por lo que da error.
- Dado un número `n`, `diasSemana[^n]` es lo mismo que `diasSemana[diasSemana.Length - n]`

```
Console.WriteLine(string.Join(", ", diasSemana));
Console.WriteLine($"La longitud del array es: {diasSemana.Length}");
Console.WriteLine($"El primer día de la semana es: {diasSemana[0]}");
Console.WriteLine($"El último día de la semana es: {diasSemana[^1]}");
```


Operador de intervalo ..

Permite definir un rango de forma sencilla.

```
Range intervalo = new Range(new Index(1), new Index(1, true));  
// equivaldrá a ...  
Range intervalo = 1..^1;
```

y por tanto el código de ejemplo que usamos en los rangos se simplificará muchísimo siendo mucho más legible y evitando usar explícitamente los tipos `Index` y `Range`.

```
string[] palabras = { "cero", "uno", "dos", "tres" };  
  
// Mostrará "uno", "dos"  
Console.WriteLine(string.Join(", ", palabras[1..^1]));
```

Ten en cuenta y recuerda que:

- Un rango especifica el inicio y el final de dicho rango de índices.
- 🙌 **Importante:** El inicio del rango es **inclusivo**, pero el final es **exclusivo**, es decir el inicio está incluido en el rango, pero el final no.

Veamos uno cuantos **ejemplos de uso** a través del **array de días de la semana** que definimos en los índices.

```
string[] diasSemana = ...  
  
string[] diasLaborales1 = diasSemana[0..5];  
Console.Write("Laborales: ");  
Console.WriteLine(string.Join(", ", diasLaborales1));  
// Fíjate que diasLaborales1.Length es 5 - 0 = 5  
  
string[] finSemana1 = diasSemana[5..7];  
Console.Write("Fin de semana: ");  
Console.WriteLine(string.Join(", ", finSemana1));  
// Fíjate que finSemana1.Length es 7 - 5 = 2  
  
string[] finSemana2 = diasSemana[^2..^0];  
Console.Write("Fin de semana: ");  
Console.WriteLine(string.Join(", ", finSemana2));  
  
string[] diasLaborales2 = diasSemana[..5];  
Console.Write("Laborales: ");  
Console.WriteLine(string.Join(", ", diasLaborales2));  
  
string[] finSemana3 = diasSemana[5..];  
Console.Write("Fin de semana: ");  
Console.WriteLine(string.Join(", ", finSemana3));
```

Además, **se pueden usar variables** para los índices y rangos:

```
Index ultimoDiaSemana = ^1;
Index primerDiaSemana = 0;
Console.WriteLine($"El primer día de la semana es: {diasSemana[primerDiaSemana]}");
Console.WriteLine($"El último día de la semana es: {diasSemana[ultimoDiaSemana]}");

string[] todosDiasSemana = diasSemana[primerDiaSemana..];
Console.WriteLine(string.Join(", ", todosDiasSemana));

Range diasFinSemana = 5..;
string[] finSemana4 = diasSemana[diasFinSemana];
Console.WriteLine("Fin de semana: ");
Console.WriteLine(string.Join(", ", finSemana4));
```



Ideas de manejo de arrays con C# modernas similares

- Fíjate que los intervalos `[0..5]` y `[5..7]` son consecutivos y disjuntos
- Fíjate que `[n..^n]` elimina n elementos de cada extremo
- Fíjate que `[..n]` es lo mismo que `[0..n]` y que `[n..]` es lo mismo que `[n..0]`
- Fíjate que podemos deducir una forma muy simple de copiar un array de forma simple utilizando el rango `[..]`

En lugar de hacer el **obsoleto**:

```
string[] v = [ "be", "eat", "see" ];
string[] copiaV = new string[v.Length];
Array.Copy(v, copiaV, copiaV.Length);
```

Ahora podremos hacer una copia haciendo simplemente:

```
string[] v = [ "be", "eat", "see" ];
// El rango de creación del subarray en todo el array de origen.
string[] copiaV1 = v[..]; // Operador de rango
string[] copiaV2 = [..v]; // Operador elemento de propagación
```

Además, todo lo visto con arrays, **también funciona con cadenas**.

```
string cadena = "Hola, mundo";
Console.WriteLine($"Primer carácter: {cadena[0]}");
Console.WriteLine($"Último carácter: {cadena[cadena.Length - 1]}");
Console.WriteLine($"Último carácter: {cadena[^1]}");
Console.WriteLine($"Primera palabra: {cadena[0..4]}");
Console.WriteLine($"Última palabra : {cadena[^5..]}");
Console.WriteLine($"Toda la cadena : {cadena[..]}");
```



Nota: En la [documentación oficial](#) podrás encontrar más información sobre el tema.

🎓 Caso de estudio:

Supongamos el siguiente código donde tenemos un array de nombres de alumnos...

```
string[] nombres =
{
    "Ana", "Pepe", "Juan", "Carmen", "Simon", "Emy", "Juanjo", "Xusa",
    "Cristina", "Jose", "Mario", "Candela", "Soledad", "Felipe", "Miguel", "Manuel"
};

Random semilla = new Random();
```

y queremos mostrar grupos de 3 o 4 alumnos consecutivos aleatoriamente, más un último grupo con los que nos queden. ¿Se te ocurre cómo solucionarlo **usando intervalos**?

En un primer caso podemos utilizar un bucle anidado o un método auxiliar para rellenar el array de componentes del grupo.

```
int i = 0;
while (i < nombres.Length)
{
    int componentesGrupo = semilla.Next(3, 5);
    componentesGrupo = i + componentesGrupo >= nombres.Length
        ? nombres.Length - i
        : componentesGrupo;

    // -----
    string[] grupo = new string[componentesGrupo];
    for (int j = 0; j < componentesGrupo; j++)
    {
        grupo[j] = nombres[i+j];
    }
    // -----
    Console.WriteLine(string.Join(", ", grupo));
    i += componentesGrupo;
}
```

Otra opción sería copiar de una sola vez al nuevo array dimensionado.

```
...
// -----
string[] grupo = new string[componentesGrupo];
Array.Copy(nombres, i, grupo, 0, componentesGrupo);
// -----
...
```

Por último, podemos extraer directamente un intervalo.

```
...
// -----
Console.WriteLine(string.Join(", ", nombres[i..(i + componentesGrupo)]));
// -----
...
```

Caso de estudio:

Veamos un ejemplo de uso de combinación de arrays en C# moderno en contraposición a otro lenguaje como JavaScript y así darnos cuenta de la convergencia en sintaxis y funcionalidades de los lenguajes modernos.

En las versiones anteriores del lenguaje C# tendríamos que hacer algo como lo siguiente para combinar los primeros `n` elementos de un array `v1` con los últimos `n` de otro array `v2`.

```
string[] v1 = new string[] { "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" };
string[] v2 = new string[] { "1", "2", "3", "4", "5", "6", "7", "8", "9", "10" };
int n = 4;

var v3 = new string[n + n];
Array.Copy(v1, 0, v3, 0, n);
Array.Copy(v2, v2.Length - n, v3, n, n);

Console.WriteLine(string.Join(" ", v3));
```

En cambio, en **C# 11**, podemos hacerlo de una forma más sencilla y legible.

```
string[] v1 = [ "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" ];
string[] v2 = [ "1", "2", "3", "4", "5", "6", "7", "8", "9", "10" ];
int n = 4;

string[] v3 = [ ..v1[..n], ..v2[^n..] ];

Console.WriteLine(string.Join(" ", v3));
```

Fijate que es similar a la forma de hacerlo en **JavaScript**.

```
let v1 = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]
let v2 = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
let n = 4

let v3 = [...v1.slice(0, n), ...v2.slice(-n)]

console.log(v3.join(" "))
```

👉 Operadores condicionales Null ?. y ?[]

Ahora que hemos empezado a usar objetos definidos en el lenguaje, tenemos que saber que uno de los errores más temidos por los programadores en lenguajes como Java o C#, es tener un **tipo referencia anulable** que apunte a Null e intentar acceder a una propiedad o realizar una operación sobre dicho objeto.

Obtendremos un error en tiempo de ejecución de `NullReferenceException` aunque posiblemente durante la compilación obtengamos un **aviso** al ser el tipo anulable.

🔴 **Nota:** Este aviso no existía en las versiones antiguas de C# al no existir los tipos referencia anulables y podíamos obtener errores de este tipo en producción. Tampoco existe en lenguajes como Java.

Este tipo de error es precisamente lo que tratamos de evitar con los tipos no anulables en los lenguajes modernos y con operadores como `??` o `?.`

Veamos unos ejemplos:

```
string? t = default; // Inicializamos a null
// ...
// Durante el código nos olvidamos de asignarle un objeto cadena y sigue a null.
// ...
Console.WriteLine(t.ToUpper()); // NullReferenceException al ejecutar.
```

👉 Una posible solución sería usar un operador ternario si existe en el lenguaje.

```
string? t = default;
Console.WriteLine(t != null ? t.ToUpper() : "");
```

👍 👍 Una mejor solución aún será usar el **operador** `?.` pues permite controlar el acceso encadenado `a?.b?.c?.d`

```
string? t = default;
Console.WriteLine(t?.ToUpper() ?? "");
```

💀 También nos puede pasar con objetos dentro de colecciones.

```
string?[] saludos = {"Hola", null, "Adios"};

// En la segunda cadena obtendremos un NullReferenceException
foreach (string? saludo in saludos)
    Console.WriteLine(saludo.ToUpper());
// Una posible solución es combinar una vez más el operador ?. con ??
foreach (string? saludo in saludos)
    Console.WriteLine(saludo?.ToUpper() ?? "No hay dato");
```

💀 Pero el operador `[]?` entonces. ¿Cuándo se usaría?. En el ejemplo anterior además de los objetos que contiene el array de cadenas, es el propio array de cadenas el que puede estar sin inicializar. Imaginemos el siguiente código ...

```
string?[] saludos = default;
...
// Tanto el array como el contenido del array podrían estar a Null.
string? t = saludos[1].ToUpper();
// Con operadores ternarios quedaría bastante ilegible y ofuscado
// incluso indentado el código.
string? t = saludos != null
    ? saludos[1] != null
        ? saludos[1].ToUpper()
        : null
    : null;
// El código equivalente al anterior usando []? y ?. sería...
string? t = saludos?[1]?.ToUpper();
```

Operador supresión de NULL .! ('null-forgiving')

Es un operador que podemos encontrar en el contexto de algunos lenguajes con control de nulos como C# (o! y o!.) siendo o una referencia a un **objeto anulable**. En **TypeScript** sería el mismo o en **Kotlin** sería (o!! y o!!.).

Es un operador que se usa de forma conjunta o en contraposición con o?.

Supongamos el siguiente código donde permitimos la posibilidad de que una colección contenga objetos anulables ...

```
static void Main()
{
    string?[] textos = { "hola", null, "adios" };
    foreach (string? t in textos)
    {
        // El compilador nos mostrará un warning indicando
        // que t puede ser null y sucederá para el segundo elemento.
        Console.WriteLine($"Texto = {t}, Logitud = {t.Length}");
    }
}
```

Sin embargo, si *'estamos seguros'* de que el array no va a contener ningún null. Podemos usar el operador de supresión de Null. Nos quitará el Warning, haciendo ver al compilador de que somos concientes de que el objeto no es nulo y si lo es debería saltar una excepción.

```
foreach (string? t in textos)
{
    Console.WriteLine($"Texto = {t!}, Logitud = {t!.Length}");
}
```

En contraposición, si no estamos seguros de que el objeto no sea nulo, deberemos usar los operadores ya vistos ?. y/o ??.

```
foreach (string? t in textos)
{
    Console.WriteLine($"Texto = {t ?? "Nulo"}, Logitud = {t?.Length ?? 0}");
}
```

En el código anterior desaparecerá el Warning y para el segundo elemento mostrará.

```
Texto = Nulo, Logitud = 0
```

Inferencia de tipos al instanciar un objeto

Al principio del tema vimos que para instanciar un objeto debemos indicar el tipo en el momento de la construcción.

```
Cuenta cuenta = new Cuenta();
```

Sin embargo el lenguaje C#, al igual que otros fuertemente tipados como Java o Kotlin, dispone de la palabra reservada `var` que usaremos en lugar de un tipo concreto al definir una variable. De esta manera indicaremos al compilador que **deduzca el tipo de la variable** a partir del tipo al que se evalúe la expresión que se encuentra en el lado derecho.

👉 **Importante:** Una vez deducido y asignado el tipo a la variable, esta no podrá cambiar y por tanto ya no le podremos asignar ningún valor que se evalúe a otro tipo.

```
var cuenta = new Cuenta(); // cuenta es compilada como tipo Cuenta

// Otros ejemplos
var i = 5; // i es compilada como entero
var s = "Hola"; // s como string
var a = new[] { 0, 1, 2 }; // a como int[] Sin embargo var a = { 0, 1, 2 }; no sería correcto.
```

También podemos usar la **expresiones de tipo de destino con** `new`. Incluidas en C# 9 me permiten hacer algo similar, pero en lugar de no poner el tipo de la variable, nos ahorraremos poner el nombre del constructor.

```
Cuenta cuenta = new Cuenta();

// Podremos hacer ...
Cuenta cuenta = new();

// o por ejemplo
String separacion = new('-', 80);

// En este caso en el AppendLine deduce que estoy creando una cadena
// ya que es uno de los tipos que espera como parámetro dicho método.
StringBuilder s = new("Hola");
s.AppendLine(new(' ', 10));
Console.Write(s);
```

🔴 **Nota:** Algunos autores consideran esta práctica preferible al uso de var.

¿Cuándo usar estas inferencias de tipos?

1. Cuando el tipo sea obvio.

```
var s = "Hola"; // Queda claro que s es un String
```

2. Para **simplificar** en los casos en que el tipo sea muy largo o enrevesado.

```
TipoConUnIdentificadorExcesivamenteLargo d = new TipoConUnIdentificadorExcesivamenteLargo();

// Puede escribirse como ...
TipoConUnIdentificadorExcesivamenteLargo d = new(); // Mejor práctica.
var d = new TipoConUnIdentificadorExcesivamenteLargo();
```

3. Porque existirán unos objetos especiales sin definición de clase que tendrán un **tipo anónimo**, y por tanto no lo podremos poner en tiempo de compilación. (Ya los veremos más adelante en el curso).