

Ferrando Eduardo – Ferrando Matías



# INTRODUCCIÓN AL LENGUAJE C#

Apunte de Catedra

# Contenido

GLOSARIO .....	5
IMPLEMENTACIONES DE .NET .....	6
¿Qué es .NET? .....	6
.NET Standard .....	7
.NET Core .....	8
NET Framework .....	8
¿Qué es C# (Sharp)? .....	8
Cómo está formado el framework .NET .....	8
Creacion de un proyecto en Visual Studio.....	9
GENERALIDADES .....	12
Comentarios .....	14
Sentencias.....	14
Bloques de código .....	15
GESTION DE DATOS .....	15
Variables .....	15
Constantes .....	16
TIPOS DE DATOS .....	17
Tipos de datos básicos .....	19
Numéricos.....	20
Caracteres.....	20
Caracteres (character) representa un carácter o dígito (char).....	20
Cadena (String) Representa una cadena de caracteres .....	20
Lógicos .....	20
Lógico (boolean) Representa un valor falso o verdadero (bool).....	20
De Objeto.....	20
Definidos por el usuario .....	21
Palabras reservadas.....	21
CONVERSIONES DE TIPOS DE DATOS .....	22
Las Conversiones pueden ser implícitas o explícitas .....	23
Conversiones implícitas: .....	23
Conversiones explícitas (conversiones de tipos): .....	24
AMBITO DE VARIABLES.....	26
▪ A nivel de bloque: .....	26
▪ A nivel de función: .....	26

▪ A nivel de clase: .....	26
MODIFICADORES DE ACCESO .....	26
▪ public .....	26
▪ protected .....	26
▪ internal .....	26
▪ protected .....	26
▪ private.....	26
OPERADORES.....	27
Operadores Aritméticos .....	27
Operadores de Asignación.....	28
Operadores relacionales.....	29
Operadores lógicos.....	30
Operaciones con cadenas.....	32
Operador condicional .....	32
INSTRUCCIONES DE CONTROL DE FLUJO.....	32
Herramienta de Decisión Simple – IF – ELSE – ELSE IF .....	33
Herramienta de Decisión Múltiple (SWITCH) .....	35
Uso de Switch .....	35
Ciclos de repetición .....	38
For.....	38
Foreach .....	42
Bucles.....	44
Instrucción Break.....	47
Instrucción Continue .....	47
Instrucción Return .....	48
Instrucción Goto .....	49
ESTRUCTURAS DE DATOS .....	50
VECTOR / ARRAY / MATRICES / ARREGLOS .....	50
Recorrer Vectores.....	52
Array Implícito .....	53
Propiedad Length .....	53
La Clase Array .....	53
Matrices o vectores multidimensionales .....	58
Colecciones.....	58
List <T>.....	59
LinkedList.....	62

Queue (Listas tipo cola) .....	63
Stack (Listas tipo Pila) .....	63
Diferencias de rendimiento entre List, matrices y Array Class en C#.....	64
MANEJO DE EXCEPCIONES (Control de Errores) .....	69
Métodos – Funciones y Procedimientos .....	70
Introducción .....	70
Funciones – Retornan un valor.....	71
Procedimientos – No retornan valor .....	72
Parámetros .....	72
Pasar parámetros por valor .....	72
Pasar parámetros por referencia.....	74
Clase Math en C#.....	79
Método Max .....	79
Método Min.....	80
Método Pow .....	80
Método sqrt.....	81
Método abs.....	81
Método Round.....	81
Clase String .....	82
CompareTo .....	83
Concat.....	83
Contains.....	83
EndsWith .....	83
Equals .....	84
IndexOf .....	84
Insert.....	84
IsNullOrEmpty.....	84
IsNullOrWhiteSpace.....	84
LastIndexOf.....	85
Lenght .....	85
Remove.....	85
Replace .....	85
Split.....	85
StartsWith.....	86
Substring.....	86
ToLower .....	86

ToUpper.....	86
Trim.....	86
TrimStart y TrimEnd.....	87
ToString.....	87

# GLOSARIO

- ❖ **.NET** Es un amplio conjunto de bibliotecas de desarrollo multilenguaje y multiplataforma que está pensado para correr sobre distintos entornos operativos, y que pretende unificar los distintos paradigmas de la programación (escritorio, cliente-servidor, WEB, APIs, Datos, Business Intelligence, etc.) en un mismo ambiente de trabajo.
- ❖ **.NET Framework** es un entorno de ejecución administrado para Windows que proporciona diversos servicios a las aplicaciones en ejecución. Soporta múltiples lenguajes de programación. Está compuesto por:
  - CLR (Common Language Runtime): motor que controla las aplicaciones en ejecución.
  - Lenguajes de compilación
  - Biblioteca de clases de .Net: código probado y reutilizable
- ❖ **C#**: Lenguaje de programación orientado a objetos que permite crear aplicaciones que se ejecutan en el entorno .NET Framework.
- ❖ **Compilador**: Código ejecutable que traduce un código escrito en un lenguaje de alto nivel como C# a lenguaje máquina.
- ❖ **Framework** Es un ambiente o entorno de trabajo. Está compuesto por un conjunto de recursos que simplifican la etapa de desarrollo, como por ejemplo: bibliotecas y funcionalidades ya programadas que sirven de base para la programación de un sistema. Provee una estructura y una especial metodología de trabajo.
- ❖ **API (Interfaz de programación de aplicaciones)**: Biblioteca formada por subrutinas, procedimientos y funciones que se puede utilizar para desarrollar otros programas.
- ❖ **ASP.NET**: Framework comercializado por Microsoft para programar sitios web dinámicos, aplicaciones web y servicios web.
- ❖ **Función**: Bloque de código que realiza una determinada operación. Al llamar a una función, se le pueden pasar parámetros de entrada. Puede devolver un valor de salida.
- ❖ **GUI (Interfaz gráfica de usuario)**: Son las pantallas mediante las que el usuario puede interactuar con el sistema.
- ❖ **Implementación**: Instalación y puesta en marcha de un software.
- ❖ **Inicialización**: Asignación de un valor inicial a una variable.
- ❖ **Iteración**: Ejecución repetitiva de instrucciones hasta que se cumple una condición (ejecución de un bucle).
- ❖ **Microsoft Visual Studio**: Es un entorno de desarrollo integrado (IDE) que tiene un editor de texto donde se puede escribir el código de un programa. También permite depurar y compilar el código para luego publicar una aplicación.

- ❖ **Operador:** Símbolo que determina el tipo de cálculo que se realizará en una expresión. Hay operadores lógicos, aritméticos, de concatenación, de comparación y especiales.
- ❖ **Procedimiento:** Grupo de instrucciones agrupadas bajo un mismo nombre. El procedimiento recibe datos y devuelve resultados.
- ❖ **Tipo de dato:** Atributo de los datos que indica sobre qué clase de dato va a trabajar el código (procedimiento, función, método, instancia, clase, proceso iterativo, etc). El tipo de dato determina qué valores puede tomar la variable y qué operaciones se podrán realizar con esos valores. Algunos tipos de datos son: expresiones booleanas, caracteres de texto, números enteros, decimales y otros.
- ❖ **Variable:** Espacio en memoria a la que se le asocia un nombre para almacenar un valor. El nombre permite referenciar la variable para realizar operaciones lógicas, matemáticas y otras (según el tipo de dato que almacenen). El valor puede ser asignado, modificado y también reasignado.

## IMPLEMENTACIONES DE .NET

### ¿Qué es .NET?

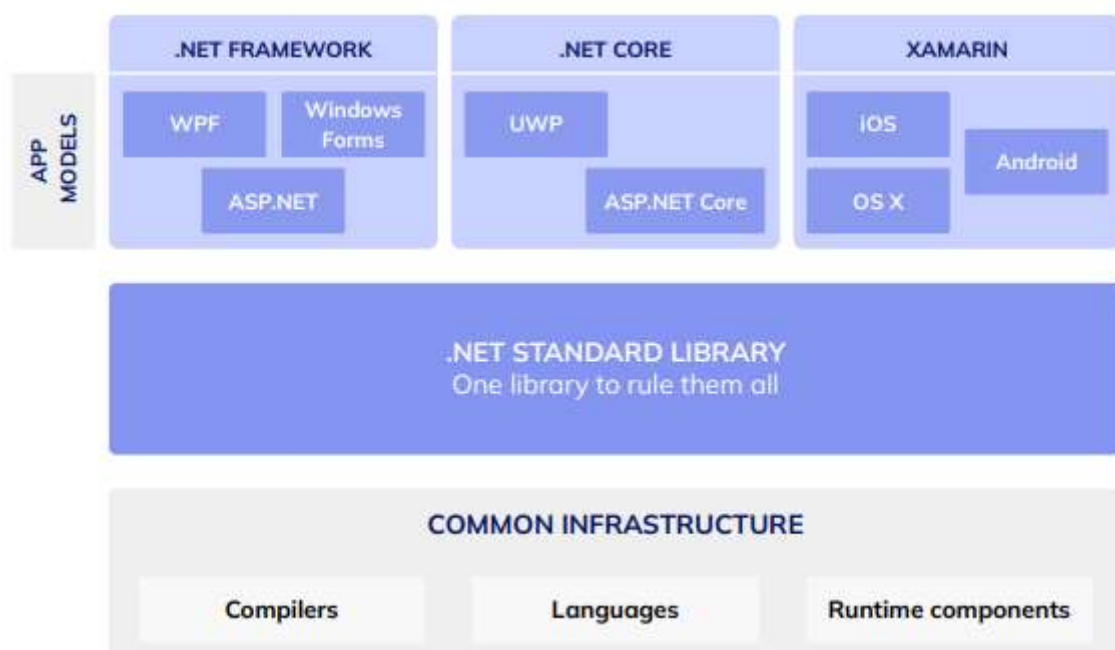
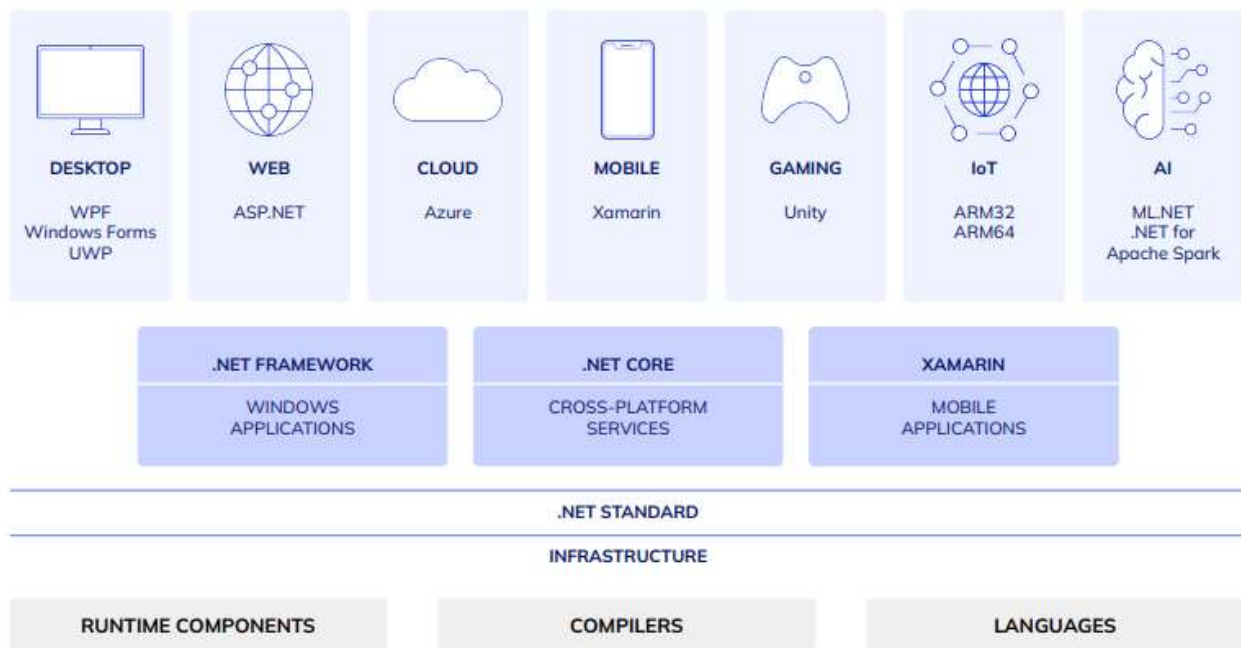
Es un amplio conjunto de bibliotecas de desarrollo multilenguaje y multiplataforma que está pensado para correr sobre distintos entornos operativos, y que pretende unificar los distintos paradigmas de la programación (escritorio, cliente-servidor, WEB, APIs, Datos, Business Intelligence, etc.) en un mismo ambiente de trabajo.

.NET está formado por cuatro componentes principales, a saber:

1. **El CLR (Common Language Runtime).** Es el entorno de ejecución para las aplicaciones codificadas en alguno de los diferentes lenguajes que cumplen con los requisitos sintácticos y semánticos de .NET.
2. **El MSIL (Microsoft Intermediate Language).** Representa el código objeto al que es transformado el código fuente escrito en cualquiera de los lenguajes aceptados por .NET. Este código fuente luego va a ser compilado a código nativo por el compilador JIT (Just In Time) que provee el CLR en tiempo de ejecución.
3. **La biblioteca de clases .NET Framework.** Es una extensísima colección de clases base (plantillas), funcionales para todos los propósitos imaginables, que es implementada y usada por todos los lenguajes e IDE's que corren sobre .NET.
4. **El entorno de desarrollo integrado (IDE).** Es el ambiente de trabajo propiamente dicho conformado por herramientas de diseño, editores, depuradores y diversos asistentes, para que el programador pueda diseñar, codificar y testear sus aplicaciones. Existen varios (Sharp Develop, RAD Studio, etc.), pero nosotros usaremos el propio de Microsoft: Visual Studio.

## .NET Standard

.NET Standard es un conjunto de APIs que se implementa mediante la biblioteca de clases base de una implementación de .NET. Más formalmente, es una especificación de APIs de .NET que constituye un conjunto uniforme de contratos contra los que se compila el código. Estos contratos se implementan en cada implementación de .NET. Esto permite la portabilidad entre diferentes implementaciones de .NET, de forma que el código se puede ejecutar en cualquier parte; por ejemplo, esto logra que la implementación .NET Core sea multiplataforma.





## ***.NET Core***

.NET Core es una implementación multiplataforma de .NET diseñada para aplicaciones instaladas tanto en servidores propios como servidores en la nube a gran escala. .NET Core puede ejecutarse en entornos Windows, macOS y Linux. Implementa .NET Standard, de forma que cualquier código que tenga como destino .NET Standard se pueda ejecutar en .NET Core.

## ***.NET Framework***

.NET Framework es la implementación original de .NET que existe desde 2002. Es el mismo entorno que los desarrolladores de .NET han usado siempre. Las versiones 4.5 y posteriores implementan .NET Standard, de forma que el código que tiene como destino .NET Standard se puede ejecutar en esas versiones de .NET Framework. Contiene API's específicas de Windows adicionales, como la API para el desarrollo de aplicaciones de escritorio con Windows Forms y WPF. .NET Framework está optimizado para crear aplicaciones de escritorio de Windows.

## ***¿Qué es C# (Sharp)?***

C# (se pronuncia “se sharp”) es un lenguaje que pretende reunir lo mejor de los diversos lenguajes que compilan a nativo (C / C++, Object Pascal, ADA, etc.) y de los interpretados (Java, Perl, etc.) en uno solo, y que, además, pueda correr sobre diversos sistemas operativos. Al respecto, existe un proyecto paralelo al desarrollo de .NET de Microsoft, que se denomina “Mono” que transportó el lenguaje C# y gran parte del framework .NET a Unix / Linux / Solaris.

C# toma gran parte de la sintaxis de C / C++ y muchas de las características de Eiffel (un lenguaje de laboratorio que ideó Bertrand Meyer), ofreciendo al desarrollador un potente lenguaje 100% orientado a objetos. Si bien el CLR acepta muchos otros lenguajes, el framework .NET está 100% escrito en C#; por lo tanto, es el lenguaje base para .Net.

## ***Cómo está formado el framework .NET8***

La arquitectura del framework .NET, es la implementación del CLR (Common Language Runtime) por parte de Microsoft para los lenguajes C#, Visual Basic, J#, ASP, y Javascript, más varios paquetes que dan soporte a interfaces de usuario, acceso a datos, XML y WEB agrupadas en una Librería de Clase Base (BCL) que está formada por cientos de tipos de datos.

A través de las clases suministradas en ella es posible desarrollar cualquier tipo de aplicación, desde las tradicionales aplicaciones de ventanas, consola o servicio de Windows NT hasta los novedosos servicios Web y páginas ASP.NET.

Dada su amplitud, ha sido necesario organizar las clases, incluidas en ella, en espacios de nombres que agrupen clases con funcionalidades similares. Esto es una clasificación.

Veamos los espacios de nombres más usados en la próxima tabla:

Espacio de nombres	Utilidad de los tipos de datos que contiene
System	Tipos muy frecuentemente usados, como los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura ADO.NET.
System.IO	Manipulación de archivos y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos de ejecución.
System.Web.UI.WebControls	Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Windows.Forms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.XML	Acceso a datos en formato XML.

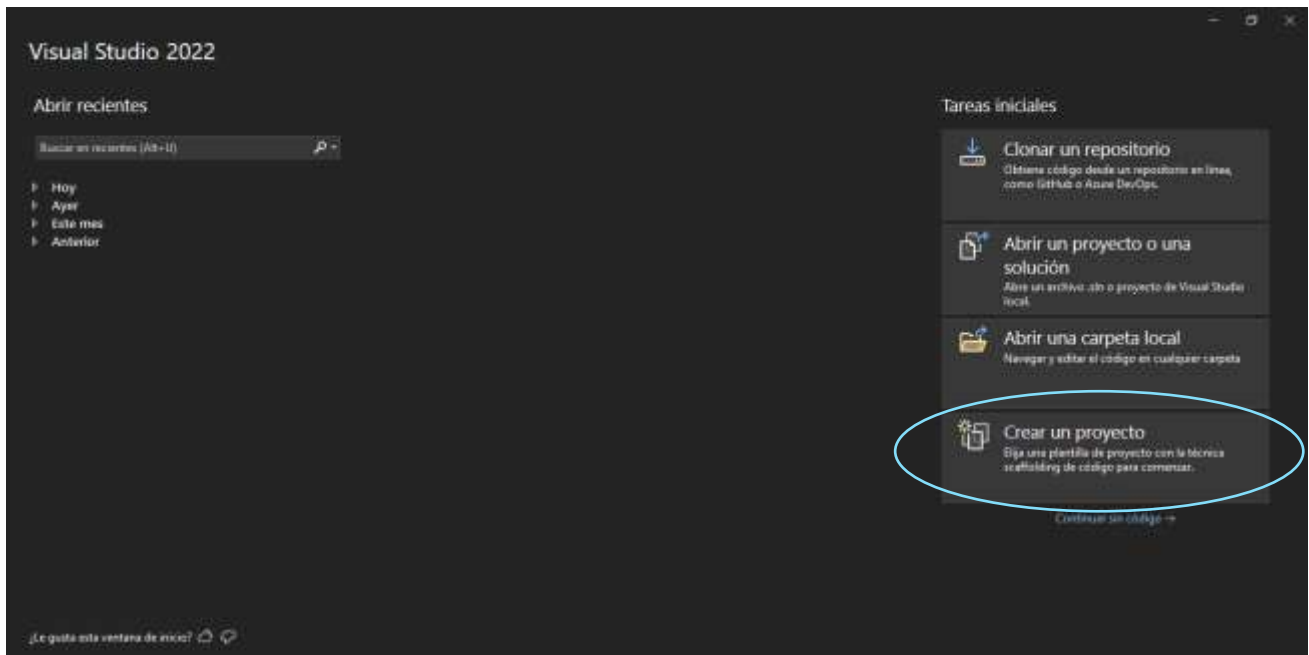
## Creacion de un proyecto en Visual Studio

Paso 1: Abrir Visual Studio 2022

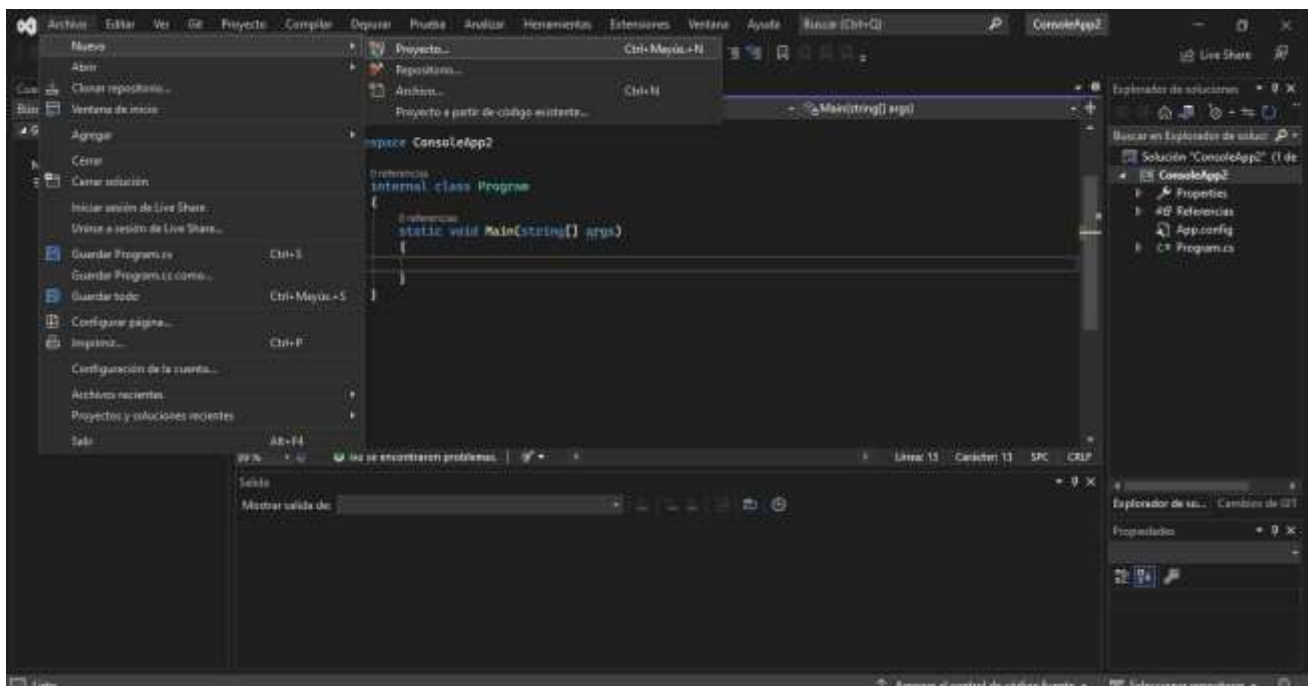
Abre Visual Studio 2022 en tu computadora. Puedes descargarlo e instalarlo desde el sitio web oficial de Microsoft si aún no lo tienes instalado (versión Community).

Paso 2: Crear un nuevo proyecto

Si es la primera vez que abres el programa seleccionas “Crear un Proyecto”.

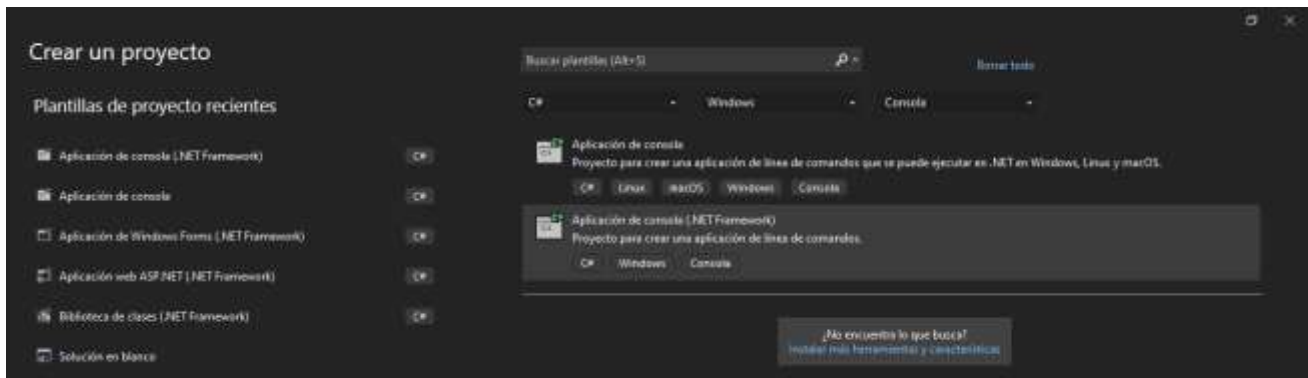
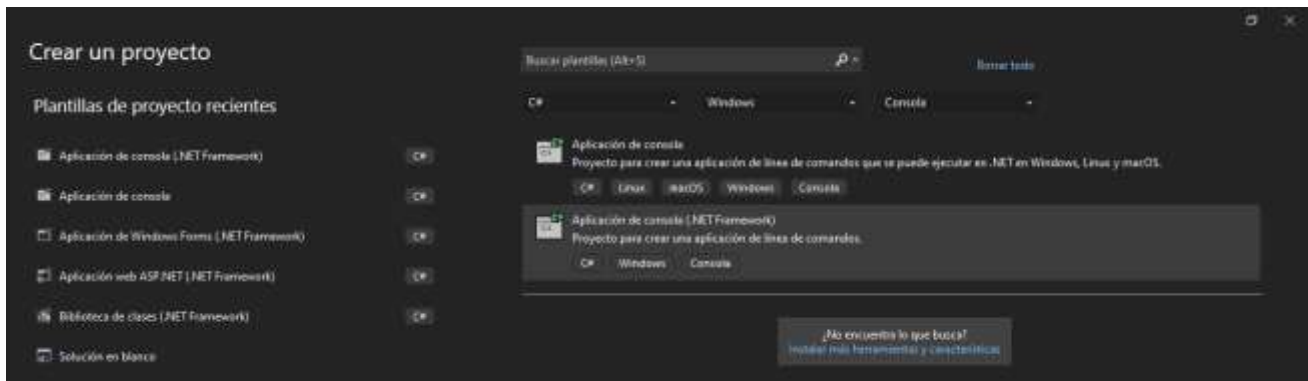


Si ya estás en un proyecto abierto haz clic en "Archivo" en la barra de menú superior y selecciona "Nuevo" > "Proyecto" para crear un nuevo proyecto en Visual Studio.



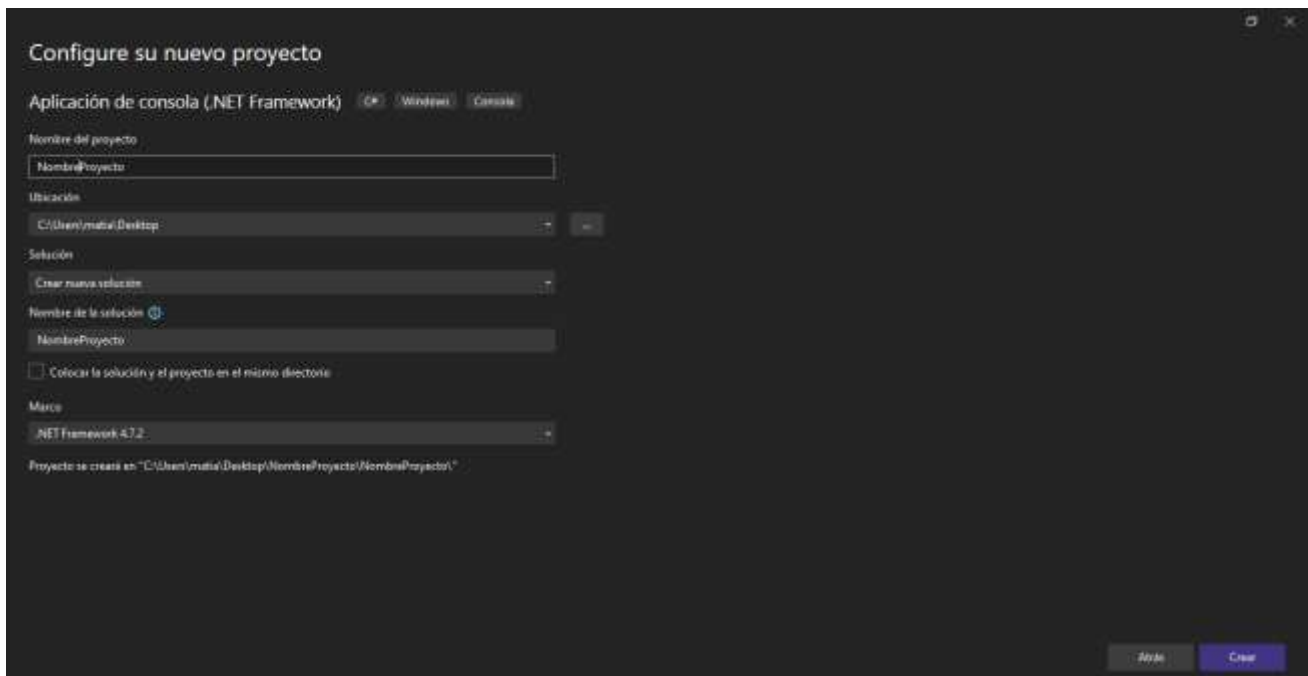
### Paso 3: Seleccionar el tipo de proyecto

En el cuadro de diálogo "Nuevo Proyecto", selecciona "Aplicación de Consola (.NET Framework)" o "Aplicación de Windows Forms (.NET Framework)" en la lista de plantillas de proyectos de C#. recuerden que pueden filtrar la búsqueda por lenguaje, plataforma y tipo de proyecto.

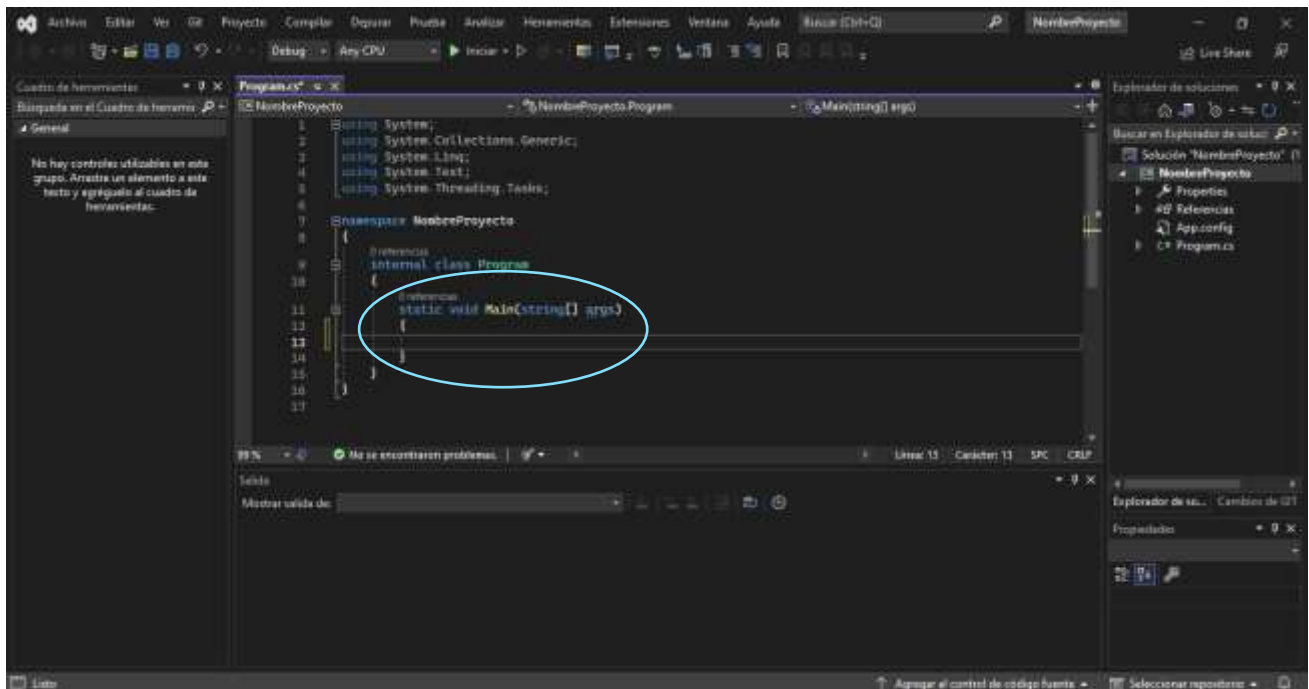


#### Paso 4: Configurar el proyecto

Asigna un nombre y una ubicación a tu proyecto en los campos "Nombre del proyecto" y "Ubicación" respectivamente. Luego, haz clic en el botón "Crear" para continuar.



En este punto ya les abra creado el nuevo proyecto de consola y podrán comenzar a codificar. Se debe tener en cuenta que lo que se ejecuta al momento de iniciar la aplicación, en tiempo de ejecución, es el método Main, por lo que será el lugar en el cual deberemos codificar nuestra aplicación.



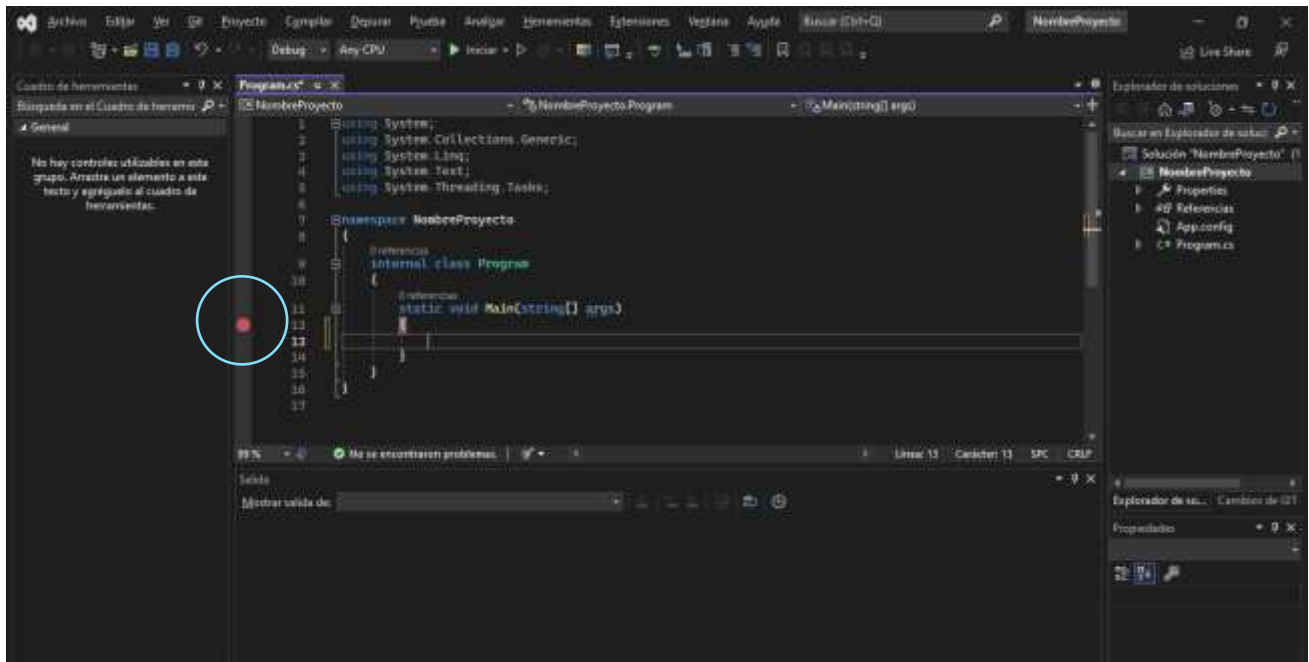
## GENERALIDADES

La mayoría de las instrucciones en C# llevan un punto y coma al final de la instrucción, salvo el caso de las estructuras (como el If, Switch, For, ForEach, While, Do While) que llevan apertura y cierre de llaves {}.

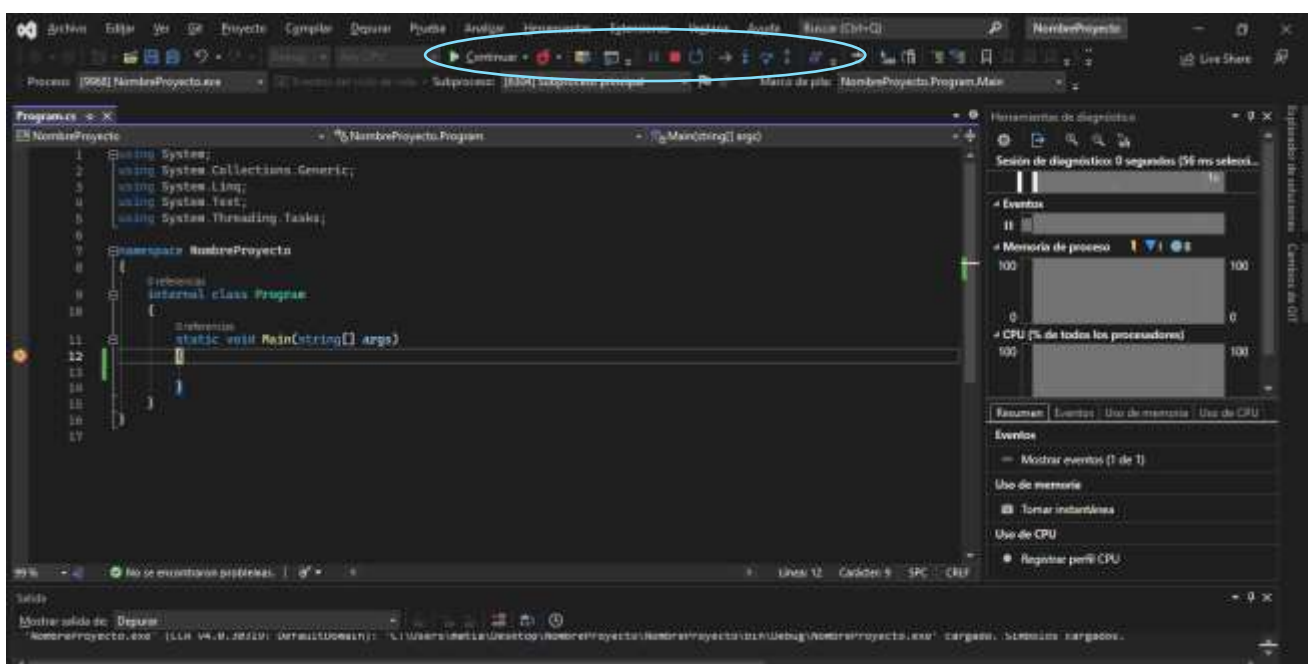
Al trabajar con la consola de windos hay distintos comandos básicos a tener en cuenta:

- `Console.WriteLine` ("Mensaje que se mostrará en la consola"); // imprime un mensaje que colocaremos entre las comillas dobles, al mismo se le puede concatenar otras cadenas o valores de variables utilizando el operador "+".
- `Console.ReadLine()`; //Lee aquello que el usuario escribió en la consola para, por ejemplo, guardarlo en una variable. Se debe tener en cuenta que todo aquello que ingrese el usuario siempre será tomado como una cadena de caracteres, por lo que para su almacenamiento en otro tipo de dato deberá ser previamente convertido. Un ejemplo sería el siguiente caso:
  - `Console.WriteLine("Ingrese su nombre");`  
`String nombre = Console.ReadLine();`
- `Console.ReadKey()`; // Espera a que el usuario presione una tecla para continuar, normalmente la utilizamos al finalizar el código ya que, dependiendo de la configuración de la consola, se nos cerrará automáticamente posterior a la ultima línea, no dejándonos visualizar los resultados.
- `Console.Clear()`; //Limpia la consola, esto nos servirá para optimizar la experiencia del usuario al utilizar el programa.

Se pueden utilizar puntos de interrupción en el programa, los cuales se colocarán realizando un click en la barra vertical a la izquierda del código, donde lo verán como un punto rojo y será ese el momento donde la ejecución del programa se detendrá para que podamos ver paso a paso que es lo que va a suceder en la aplicación y de esa manera encontrar posibles errores.



Al momento de detenerse la ejecución del programa, tendremos dos botones para poder recorrer las instrucciones (utilizando F11) o por procedimientos (utilizando F10). También encontraremos el botón para detener la ejecución y para continuar con el programa en ejecución sin analizar la depuración por instrucciones o procedimientos.



## Comentarios

Los comentarios dentro de un código, son textos aclaratorios que escribimos dentro del código y que el compilador no lo tendrá en cuenta, es decir, ese texto no afecta al programa, son sumamente útiles ya sea cuando otro programador deba interpretar nuestro código, o simplemente recordar que hace determinada función o método que olvidamos con el paso del tiempo. Existen dos maneras de indicar que texto será un comentario en C#: comentario de una sola línea, o comentario de bloque. El comentario de una línea, empezando por "//" o de varias líneas escribiendo entre "/\*" y "\*/". No hace falta poner asterisco al principio de cada línea, pero Visual Studio lo hace automáticamente y además queda mejor delimitado el comentario.

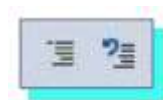
// Este sería el comentario de una línea.

/\*

\*Este sería el comentario de bloque, todo lo que se encuentre entre /\* y \*/ estará comentado  
\* y no será tomado en cuenta por el programa.

\*/

Además, el IDE de Visual Studio tiene en botones dos acciones: una para comentar un bloque de instrucciones y otra para “descomentarlo”, resultando en muchas ocasiones más ágiles.



Los comentarios en C# aparecen en el código de color verde.

## Sentencias

Una sentencia es una orden que se le da al programa para realizar una tarea específica, esta puede ser: mostrar un mensaje en la pantalla, declarar una variable (para reservar espacio en memoria), inicializarla, llamar a un método, etc. Para C# cada sentencia termina con un “;” (punto y coma). Normalmente, las sentencias se ponen unas debajo de otras, aunque sentencias cortas pueden colocarse en una misma línea. He aquí algunos ejemplos de sentencias:

- ❖ `int x = 10;`
- ❖ `using System.IO;`
- ❖ `Console.WriteLine("Hola Mundo"); //Imprimir un mensaje en consola`



En C#, los caracteres espacio en blanco se pueden emplear libremente. Es muy importante para la legibilidad de un programa la colocación de unas líneas debajo de otras empleando tabuladores. El editor del IDE nos ayudará plenamente en esta tarea sin apenas percibirlo.

## ***Bloques de código***

Un bloque de código es un grupo de sentencias que se comportan como una unidad. Se los usa para indicar el comienzo y fin de un conjunto de líneas (sentencias) de código. Por lo general, se los usa para delimitar namespaces, clases, estructuras, enumerados y métodos. Un bloque de código está limitado por las llaves de apertura "{" y cierre "}". Veamos un ejemplo:

```
static void Main() {  
    Linea1;  
    Linea2;  
}
```

Cada una de esas líneas implica una expresión que el compilador de C# analizará, verificando que su sintaxis sea la correcta. Para ello disponemos de los operadores.

# **GESTION DE DATOS**

## ***Variables***

Las variables son un espacio reservado en la memoria de la computadora, este espacio está asociado a un nombre simbólico también conocido como identificador. En este espacio podemos almacenar un valor, el cual puede ser modificado en cualquier momento durante la ejecución del programa que contiene la variable.

Para crear o declarar una variable es bastante sencillo, y se puede dividir en 3 pasos: colocar el tipo de dato que va a almacenar la variable, colocar un identificador (que sería el nombre de dicha variable) y opcionalmente asignarle un valor; al terminar, se coloca un punto y coma (;), esto indica que se terminó la instrucción.

### **Ejemplos:**

```
int nro; // variable entera sin asignar valor.  
int numero = 10; // variable entera asignándole un valor. (sin comillas).
```



```
char carácter = 'a'; // variable de un solo carácter (lleva comillas simples).
string palabra = "hola"; // variable de varios caracteres u oración (lleva comillas dobles).
double decimal = 10.5; // variable decimal (sin comillas).
```

Los identificadores deben ser nombres que hagan referencia al uso que se le dará a la variable, por ejemplo, si la variable será usada para el apellido, lo recomendado es ponerle «apellido», y evitar usar identificadores como «x», «c», «a», porque en este caso no identifican correctamente el valor de esta variable.

El identificador de **una variable debe ir en minúscula**, (esto es un estándar, no obligatorio, pero es buena práctica). También puede iniciar con una letra o guion bajo (\_), nunca números o caracteres especiales como la ñ, letras con tildes, entre otros. No se permite caracteres especiales fuera del inglés (acentos españoles, franceses, etc.). Si el identificador está compuesto por dos palabras o más se debe utilizar método **camelcase**, donde cada palabra, posterior a la primera, iniciará en mayúscula.

**Ejemplo de camelCase:** double totalSuma;

## Constantes

Las constantes que hacen que el compilador reserve un espacio de memoria para almacenar un dato, pero en este caso ese dato es siempre el mismo y no se puede modificar durante la ejecución del programa.

Un ejemplo claro sería almacenar el valor de Pi en una constante para no tener que poner el número en todas las partes donde lo podamos necesitar. Otro ejemplo, la cantidad de días de la semana, la cantidad de meses del año, etc. Se declaran de un modo similar a las variables, basta con añadir la palabra “const” en la declaración.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Circunferencia
{
    class Perimetro
    {
        static void Main(string[] args)
        {
            const double PI = 3.1415926; // Declaración de constante PI, de tipo double
            double Radio = 4; // Declaración e inicialización de constante Radio, de tipo double
            Console.WriteLine("El perímetro de una circunferencia de radio {0} es {1}", Radio, 2 * PI * Radio);
            Console.WriteLine("El área de un círculo de radio {0} es {1}", Radio, PI * Math.Pow(Radio, 2));
            Console.ReadKey();
        }
    }
}
```

# TIPOS DE DATOS

Cuando hablamos de los tipos de datos en programación nos referimos a la propiedad que determina a que grupo o conjunto pertenece un dato (dominio), también nos permite saber cómo representaremos el dato y que operaciones se pueden realizar con él.

El sistema de tipos de datos suele ser la parte más importante de cualquier lenguaje de programación. Para que una aplicación sea eficiente con el menor consumo posible de recursos, es esencial el uso correcto de los distintos tipos de datos. Muchos de los lenguajes orientados a objetos proporcionan los tipos agrupándolos de dos formas: los tipos primitivos del lenguaje, como números o cadenas, y el resto de tipos creados a partir de clases.

Esto genera muchas dificultades, ya que los tipos primitivos no son y no pueden tratarse como objetos, es decir, no se pueden derivar y no tienen nada que ver unos con otros. Sin embargo, en C# (más propiamente en .NET Framework) se cuenta con un sistema de tipos unificado, el CTS (Common Type System), que proporciona todos los tipos de datos como clases derivadas de la clase de base System.Object (incluso los literales pueden tratarse como objetos).

Es importante aclarar que: hacer que todos los datos que va a manejar un programa sean objetos puede provocar que baje el rendimiento de la aplicación. Para solventar este problema, .NET Framework divide los tipos en dos grandes grupos: los tipos valor y los tipos referencia. Cuando se declara una variable que es de un tipo valor se está reservando un espacio de memoria en la pila (stack) para que almacene los datos reales que contiene esta variable.

Por ejemplo, en la declaración:

```
int numero = 10;
```

Se está reservando un espacio de 32 bits en la pila (una variable de tipo “int” es un objeto de la clase System.Int32), en los que se almacena el 10, que es lo que vale la variable. Esto hace que la variable “num” se pueda tratar directamente como si fuera de un tipo primitivo en lugar de un objeto, mejorando notablemente el rendimiento. Como consecuencia, una variable de tipo valor nunca puede contener null (referencia nula).

Durante la ejecución de un programa, la memoria se distribuye en tres bloques: la pila (stack), el montón (heap) y la memoria global. La pila es una estructura en la que los elementos se van apilando de modo que el último elemento en entrar en la pila es el primero en salir (estructura LIFO, o sea, Last In First Out). El montón es un bloque de memoria contiguo en el cual la memoria no se reserva en un orden determinado como en la

pila, sino que se va reservando aleatoriamente según se va necesitando. Cuando el programa requiere un bloque del montón, este se sustrae y se retorna un puntero (variable que contiene direcciones de memoria) al principio del mismo. La memoria global es el resto de memoria de la máquina que no está asignada ni a la pila ni al montón, y es donde se colocan el método main y las funciones que éste invocará.

En el caso de una variable que sea de un tipo referencia, lo que se reserva es un espacio de memoria en el montón para almacenar el valor, pero lo que se devuelve internamente es una referencia al objeto, es decir, un puntero a la dirección de memoria que se ha reservado. En este caso, una variable de tipo referencia si puede contener una referencia nula (null).

¿Por qué esta “separación” de tipos y posiciones de memoria? Porque una variable de un tipo valor funcionará como un tipo primitivo siempre que sea necesario, pero podrá funcionar también como un tipo referencia, es decir como un objeto, cuando se necesite que sea un objeto. Hacer esto en otros lenguajes, como Java, es imposible, dado que los tipos primitivos en Java no son objetos.

Los tipos de datos básicos son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones, que en C# se ha incluido una sintaxis especial para tratarlos. Por ejemplo, para representar números enteros de 32 bits con signo se utiliza el tipo de dato System.Int32 definido en la BCL, aunque a la hora de crear un objeto “a”, de este tipo, que represente el valor 2, se usa la siguiente sintaxis:

```
System.Int32 a = 2;
```

Para este tipo, que es de uso muy frecuente, también se ha predefinido en C# el alias int, por lo que la definición de variable anterior queda así de compacta:

```
int a = 2;
```

System.Int32 no es el único tipo de dato básico incluido en C#. En el espacio de nombres System se han incluido:

Tipo	Descripción	Bits	Rango de valores	Alias
<b>SByte</b>	Bytes con signo	8	[-128, 127]	Sbyte
<b>Byte</b>	Bytes sin signo	8	[0, 255]	byte
<b>Int16</b>	Enteros cortos con signo	16	[-32.768, 32.767]	short
<b>UInt16</b>	Enteros cortos sin signo	16	[0, 65.535]	ushort
<b>Int32</b>	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	int
<b>UInt32</b>	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
<b>Int64</b>	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long

Tipo	Descripción	Bits	Rango de valores	Alias
<b>UInt64</b>	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	ulong
<b>Single</b>	Reales con 7 dígitos de precisión	32	[1,5×10 <sup>-45</sup> - 3,4×10 <sup>38</sup> ]	float
<b>Double</b>	Reales de 15-16 dígitos de precisión	64	[5,0×10 <sup>-324</sup> - 1,7×10 <sup>308</sup> ]	double
<b>Decimal</b>	Reales de 28-29 dígitos de precisión	128	[1,0×10 <sup>-28</sup> - 7,9×10 <sup>28</sup> ]	decimal
<b>Boolean</b>	Valores lógicos	32	true, false	bool
<b>Char</b>	Caracteres Unicode	16	['\u0000', '\uFFFF']	char
<b>String</b>	Cadenas de caracteres	Variable	El permitido por la memoria	string
<b>Object</b>	Cualquier objeto	Variable	Cualquier objeto	object

## Tipos de datos básicos

Pese a su sintaxis especial, en C# los tipos básicos son tipos del mismo nivel que cualquier otro tipo del lenguaje. Es decir, heredan de System.Object y pueden tratarse como objetos de dicha clase por cualquier método que espere un System.Object.

Esto es muy útil para el diseño de rutinas genéricas que admitan parámetros de cualquier tipo y es una ventaja importante de C# frente a lenguajes similares, como Java, donde los tipos básicos no son considerados objetos.

El valor que por defecto se les da a los campos de tipos básicos consiste en poner a cero toda el área de memoria que ocupen. Esto se traduce en que los campos de tipos básicos numéricos se inicializan por defecto con el valor 0, los de tipo bool lo hacen con false, los de tipo char con '\u0000' (carácter nulo), y los de tipo string y object con null.

Para los casos de los tipos de variables numéricas se puede especificar (durante la asignación de un valor), el tipo de dato numérico de ese valor mediante el uso de sufijos.

Los sufijos para los valores literales de los distintos tipos de datos numéricos son los siguientes:

- ❖ L (mayúscula o minúscula): long ó ulong, por este orden;
- ❖ U (mayúscula o minúscula): int ó uint, por este orden;
- ❖ UL ó LU (independientemente de que esté en mayúsculas o minúsculas): ulong;
- ❖ F (mayúscula o minúscula): single;
- ❖ D (mayúscula o minúscula): double;
- ❖ M (mayúscula o minúscula): decimal;

## Numéricos

- Entero (Integer/int): Números enteros sin coma flotante (decimales)
- Flotante (Float): Números con decimales cortos (7 decimales)
- Doble (Double): Números con decimales largos (15 decimales)
- Decimal (Decimal): Números con decimales largos (28 decimales)

## Caracteres

**Caracteres (character) representa un carácter o dígito (char)**

'#', 'r', 't', '@', '9', '8', ''

**Cadena (String) Representa una cadena de caracteres**

"Hola Mundo", "Esto es una cadena de caracteres"

## Lógicos

**Lógico (boolean) Representa un valor falso o verdadero (bool).**

True (verdadero), false (falso),

0 (false), 1 (true)

## De Objeto

El tipo de dato objeto es una estructura que nace en la programación orientada a objetos, tiene la particularidad de funcionar como un contenedor en el cual se puede colocar cualquier dato sin importar su tipo, pero con la limitante de que no es posible realizar operaciones ni ninguna instrucción que se podría realizar si el dato estuviese en una variable del mismo tipo que el dato.

Es decir, si introduzco un número entero en una variable de tipo objeto, podre almacenar y mantener el número, pero no poder realizar ni sumas, restas, multiplicación ni ninguna otra operación propia de los números.

## ***Definidos por el usuario***

Los tipos de datos ***Definidos por el usuario*** existen en algunos paradigmas de programación, como lo es el paradigma de orientación a objetos.

Estos tipos de datos tienden a ser estructuras complejas, que pueden contener más de un tipo de dato internamente e inclusive objetos, son creados a partir de prototipos o plantillas conocidas como clases y su uso es posible cuando se instancia un objeto del mismo tipo de la clase.

## **Palabras reservadas**

Las palabras reservadas no pueden ser usadas como nombre de variable. En la siguiente slide se muestran algunas de las palabras reservadas más relevantes del lenguaje C#.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
do	double	else	enum	event
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	value	int	interfaz
internal	long	new	null	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Los datos pueden ser divididos en varios grupos

- Numéricos
- Caracteres
- Lógicos
- Objetos
- Definidos por el usuario

## CONVERSIONES DE TIPOS DE DATOS

En el mundo de la programación es esencial saber trabajar con los tipos de datos, y saber cómo convertir tipos de datos, ya que en la mayoría de los casos tendremos que convertir un tipo de dato a otro, ya sea para realizar operaciones aritméticas, concatenar cadenas, redondear números, entre otros.

En estos casos tendremos que hacer uso de una técnica llamada parseo (parsing en inglés), también conocido como casting, la cual nos permite convertir un tipo de dato en otro, siempre y cuando cumpla con ciertos requisitos.

Cada lenguaje de programación tiene sus métodos para realizar la conversión de tipo, en C# tenemos la Clase Convert. Esta clase contiene los métodos necesarios para convertir a casi todos los tipos de datos existentes en C#.

Las siguientes variables contienen números, pero no todas son tipo numérico, por lo tanto, si quiero realizar una suma, solo las 3 primeras pueden ser utilizadas, las otras 3 deben ser convertidas a otro tipo de dato numérico, como pueden ser interger(int), double, o float.

```
int numero = 56; // variable tipo entero
double flotanteLargo = 56.77878m; // variable tipo flotante largo
decimal flotanteLargoDecimal = 56.7787865874d; // variable tipo flotante largo
float flotante = 2.9f; // variable tipo flotante corto
char caracter = '9'; // Variable tipo carácter
string cadena = "56"; // Variable tipo cadena
object objeto = 45.25;
```

## *Las Conversiones pueden ser implícitas o explícitas*

**Conversiones implícitas:** no se requiere una sintaxis especial porque la conversión se realiza con seguridad de tipos y no se perderán datos.

Por ejemplo, de un entero a un flotante o double.

```
int numero = 90; // numero contiene el valor 90

double n_numero; //se declara una variable de tipo double

n_numero = numero;

//se le asigna el valor de numero a n_numero, realizamos la conversion de int a double

//numero tiene como valor 90

//n_numero tiene como valor 90.0
```

También en el caso de que el tipo de dato sea Object, puede ser convertido implícitamente a otro tipo de la siguiente forma

```
float flotante = (float)objeto;

int n_entero = (int)objeto;

char n_char = (char)objeto;

string n_object = (string)objeto;

double n_doble = (double)objeto;
```



## **Conversiones explícitas (conversiones de tipos):**

Para convertir los tipos de datos tenemos dos maneras:

### ***Clase Convert***

```
int n_cadena;  
  
n_cadena = Convert.ToInt32(cadena);  
  
double n_caracter;  
  
n_caracter = Convert.ToDouble(caracter);  
  
double n_objeto;  
  
n_objeto = Convert.ToDouble(objeto);
```

### ***Método Parse***

Cada tipo de dato tiene una clase que posee un método llamado parse. Este convierte al tipo de dato desde un String:

```
string cadena = "5";  
  
float flotante = float.Parse(cadena);  
  
int n_entero = int.Parse(cadena);  
  
char n_char = char.Parse(cadena);  
  
double n_doble = double.Parse(cadena);
```

La diferencia entre la clase Convert y el método parse radica en que la primera acepta un valor nulo y devuelve 0 en su lugar, y el segundo no lo aceptará y devolverá una excepción (error). Por otra parte, Convert es más lento en cuanto a rendimiento, pero eso solo se notaría al procesar grandes cantidades de datos.

Si se necesita convertir cualquier tipo a String (cadena) solo tienes que llamar el método “.ToString()” de la siguiente forma:

```
float flotante = 45.9f;  
string n_string = flota.ToString();
```

## Método TryParse

El método TryParse es una función incorporada en C# que se utiliza para convertir una cadena de texto en un tipo de dato primitivo, como entero (int), decimal (decimal), booleano (bool) y otros tipos de datos numéricos, sin generar una excepción si la conversión falla. Esto es útil cuando se espera que la cadena de texto pueda no ser válida para la conversión y no se quiere que el programa genere una excepción en ese caso.

Sintaxis general del método TryParse:

```
bool TryParse(string valor, out tipoDato resultado)
```

Donde:

- **valor:** Es la cadena de texto que se intentará convertir.
- **resultado:** Es el parámetro de salida que contendrá el resultado de la conversión si esta es exitosa.

El valor de retorno del método TryParse es un booleano que indica si la conversión fue exitosa o no. Si la conversión es exitosa, el valor convertido se asignará al parámetro de salida resultado y el método TryParse retornará true. Si la conversión falla, el valor de resultado será el valor predeterminado del tipo de dato y el método TryParse retornará false.

Ejemplo de cómo se puede usar el método TryParse para convertir una cadena de texto en un entero:

```
string numeroTexto = "123";  
int numero;  
  
if (int.TryParse(numeroTexto, out numero))  
{  
    Console.WriteLine("La conversión fue exitosa. El número es: " + numero);  
}  
else  
{  
    Console.WriteLine("La conversión falló. La cadena de texto no es un número válido.");  
}  
  
Console.ReadKey();
```

En este ejemplo, la cadena de texto "123" se intenta convertir en un entero usando el método TryParse de la clase int. Si la conversión es exitosa, el número convertido se imprime en la consola. Si la conversión falla, se muestra un mensaje indicando que la cadena de texto no es un número válido.

# AMBITO DE VARIABLES

Denominamos ámbito de una variable a la porción de código donde podrá ser accedida dicha variable, es decir, si una variable fue declarada dentro de un bloque como un `for`, un `while`, etc, esta variable no va a poder ser accedida desde otra porción de código, ya que existe solo para el bloque de código donde fue creada.

Los diferentes ámbitos son los siguientes:

- **A nivel de bloque:** Sólo el código del bloque tendrá la posibilidad de trabajar con la variable (por ejemplo, un bucle ***while***)
- **A nivel de función:** Sólo el código de la función donde se declara la variable podrá modificar su contenido, se llama normalmente variable local.
- **A nivel de clase:** Una variable declarada en el interior de una clase es accesible al código de esta clase sin restricción y eventualmente a partir de otras porciones de código en función del nivel de acceso de la variable.

# MODIFICADORES DE ACCESO

Estos modificadores, los veremos más adelante cuando veamos el paradigma de Programación orientada a Objetos, más específicamente en encapsulación, pero los nombro para que los tengan presente.

**Nivel de acceso de las variables:**

- ***public*:** Los elementos declarados serán accesibles desde cualquier porción de código del proyecto y desde cualquier otro proyecto que haga referencia a aquel donde están declarados. No se pueden utilizar dentro de las funciones.
- ***protected*:** Se puede utilizar en el interior de una clase. Permite restringir el acceso a la variable únicamente al código de la clase y todas las clases que hereden de ella.
- ***internal*:** Serán accesibles desde el ensamblado en el cual están declarados y tampoco se pueden utilizar en el interior de una función.
- ***protected internal*:** Es el nivel de acceso de *protected* e *internal*.
- ***private*:** Restringe el acceso de la variable al módulo, a la clase o a la estructura en la cual está declarada. No se puede utilizar en el interior de un procedimiento o función.

# OPERADORES

Según MSDN un operador de programación es:

*Un operador en C# es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado. En C#, un operador es un elemento de programa que se aplica a uno o varios operandos en una expresión o instrucción. Los operadores que toman un operando, como el operador de incremento `++` o `new`, se conocen como operadores unarios. Los operadores que toman dos operandos, como los operadores aritméticos (`+`, `-`, `*`, `/`) se conocen como operadores binarios. Un operador, el operador condicional (`?:`), toma tres operandos y es el único operador ternario de C#.*

En esta cita tenemos 2 palabras destacadas: operadores y operandos.

- Los **operadores** son los elementos que usaremos para realizar ciertas operaciones tales como comparaciones, sumas, restas, incrementos, entre otros.
- Los **operandos** son los elementos sobre los cuales son aplicados los operadores como por ejemplo números, variables, entre otros.

Vamos a ver los más comunes:

- Operadores aritméticos
- Operadores de asignación
- Operadores lógicos
- Operadores relacionales u operadores de comparación
- Operadores de incremento y decremento

## Operadores Aritméticos

Son símbolos aritméticos básicos: suma (+), resta (-), multiplicación (\*), división (/) y Modulo (%), este último es el residuo de una división entera.

Estos toman valores numéricos, sean variables o valores literales, como sus operandos y dan como resultado un solo valor numérico, este resultado lo puedo almacenar en una variable si deseo utilizarlo más adelante.

```
// esto es una suma da como resultado 13
int suma = 5 + 8; // se realiza la suma y se almacena en la variable
double division = 8/9;
long multiplicacion = 45 * 45; // long es tipo entero(int) pero soporta cantidades más grandes
```

```
int resta = 5-1;
```

El operador Módulo (%) retorna el residuo de una división entera, es decir cuando dividimos dos números y hay una parte que no puede ser dividida por lo tanto esta parte sobra, en una división entera el resultado no tiene decimales.

```
int resultado;

int numero1 = 7;
int numero2 = 5;

resultado = numero1 % numero2; // el resultado será 2,
//porque 7 entre 5 es igual a 1, y sobran 2

numero1 = 17;
numero2 = 3;

resultado = numero1 % numero2; // el resultado será 2,
//porque 17 entre 3 es igual a 5, y sobran 2
```

El resultado de modular siempre será la parte que no se puede dividir, sería el sobrante o residuo de la división

## **Operadores de Asignación**

Los operadores de asignación se utilizan para asignar un nuevo valor a una variable, propiedad o evento. El operando izquierdo de una asignación debe ser una variable, un acceso a propiedad, un acceso a indizador o un acceso a evento.

Este asigna el valor del operando de la derecha al operando de la izquierda.

```
int numero;
string nombre;

numero = 45;
// se le asigna el valor 45(derecha) a la variable numero(izquierda)
nombre = "Jorge";
// se le asigna jorge a la variable nombre
numero = numero + 13;
// se le suma 13 al valor contenido en numero (45)
//y luego se le asigna el resultado a numero
```

**Existen sobrecargas del operador de asignación (=), estas acortan ciertas operaciones**

```

int numero1 = 4;
int numero2 = 5;
numero1++; // esto incrementa el valor de numero1 en una unidad
numero2--; // esto decrementa el valor de numero2 en una unidad
numero1 += 5; // esto equivale a numero1 = numero1 + 5
numero1 -= numero2; // esto equivale a numero1 = numero1 - numero2
numero1 *= 2; //esto equivale a numero1 = numero1 * numero2
numero2 /= numero1; //esto equivale a numero2 = numero2 / numero1
numero1 %= 2; //esto equivale a numero1 = numero1 % numero2

```

Si el operador ++ se coloca tras el nombre de la variable (como en el ejemplo anterior) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el operador --.

La ventaja de usar los operadores ++ y -- es que en muchas máquinas son más eficientes que otras formas de realizar sumas o restas de una unidad, pues el compilador puede traducirlos en una única instrucción en código máquina.

Por ejemplo:

```

c = b++; // Se asigna a c el valor de b y luego se incrementa b
c = ++b; // Se incrementa el valor de b y luego se asigna a c

```

**Operadores relacionales**

Son símbolos usados para comparar dos valores, si la expresión evaluada es correcta, entonces estos operadores dan como resultado verdadero, en caso contrario falso

Operador	Significado	Ejemplo
==	Igual que	a == b
!=	Distinto que	a != b
<	Menor que	a < b
>	Mayor que	a > b
<=	Menor o igual que	a <= b
>=	Mayor o igual que	a >= b

```

bool resultado;

resultado = 5 > 1; // esto da como resultado verdadero (true)
resultado = 5 <= 5; // esto da como resultado verdadero(true)
//debido a que aunque 5 no es menor que 5, si es igual a 5

String valor1 = "juan";
String valor2 = "pepe";

resultado = valor1 != valor2; // esto es verdadero, porque juan es diferente a pepe
resultado = "pepe" == valor2; // esto es como resultado verdadero, ambos valores son iguales

resultado = valor1 == "Juan"; // esto es falso (false), debido a que juan y Juan
//son valores diferentes, C# distingue entre minúsculas y mayúsculas

```

## Operadores lógicos

Estos trabajan usando la tabla de verdad:

El Operador && (AND) que quiere decir «y» expresa que si ambas expresiones son verdaderas entonces el retorna el valor verdadero.

El operador lógico && (AND)

P	Q	P && Q
V	V	V
V	F	F
F	V	F
F	F	F

El operador || (OR) que quiere decir «o» expresa que al menos una de las dos expresiones debe ser verdadera y retornara verdadero

El operador lógico || (OR)

P	Q	P    Q
V	V	V
V	F	V
F	V	V
F	F	F

El operador ! (NOT) quiere decir «no» o «negación», cambia el valor al su inverso, es decir si es verdadero, el resultado cambiara a falso, y viceversa

### El operador lógico ! (NOT)

P	!P
V	F
F	V

Los operadores && y || se diferencian de & y | en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que, si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente. La evaluación no perezosa consiste en evaluar siempre ambos operandos.

Es decir, si el primer operando de una operación && es falso, se devuelve false directamente, sin evaluar el segundo; y si el primer operando de una || es verdadero, se devuelve true directamente, sin evaluar el otro.

Ahora bien, ¿cómo utilizamos esto en programación?, la respuesta es simple, podemos combinar los operadores relacionales con los lógicos.

```
int n1 = 78;
int n2 = 90;
int n3 = 9;

bool resultado;

resultado = n1 < n2 && n2 > n3;
//en este caso el resultado es verdadero
//debido a que n1 < n2 es verdadero
//y n2 > n3 tambien es verdadero
//aplicamos el operador && ( leer operador &&)

resultado = n1 != 70 || n3 > 10;
//en este caso el resultado es verdadero
//debido a que n1 != 70 es verdadero
//n3 > 10 es falso, pero el operador OR solo necesita uno verdadero
//aplicamos el operador ||
```



```

resultado = true;
resultado = !resultado;
//en este caso el resultado es falso
//debido a que la variable "resultado" contiene el valor true(verdadero)
//pero aplicando el operador de negación NOT(!), hemos invertido el valor

//en este caso usamos el if para aprovechar los operadores
if(n1>10){
    Console.WriteLine("Es Mayor"); //por consola
    MessageBox.Show("Es mayor"); //Por ventana
}

```

Tanto los operadores relacionales y los lógicos son utilizados por las estructuras de control de flujo (if, while, for)

## **Operaciones con cadenas**

Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos.

Por ejemplo, "Hola" + "mundo" devuelve "Hola mundo", y "Hola mund" + 'o' también.

## **Operador condicional**

Es el único operador incluido en C# que toma 3 operandos, y se usa así:

```
<condición> ? <expresion1> : <expresion2>
```

Esto es, se evalúa la condición. Si es cierta, se devuelve el resultado de expresion1, y si es falsa se devuelve el resultado de expresion2.

# **INSTRUCCIONES DE CONTROL DE FLUJO**

Como se ha visto hasta ahora, los programas ejecutan las líneas de código en orden secuencial, una instrucción detrás de otra. Sin embargo, hay muchas situaciones en las que es preciso alterar ese orden, o bien puede ocurrir que sea necesario que se efectúen una serie de operaciones que pueden ser distintas en otras circunstancias.

Por ejemplo, si el programa pide una clave de acceso de un usuario, deberá continuar con la ejecución normal en caso de que la clave introducida sea correcta, y deberá salir del mismo en caso contrario.

Pues bien: para todas estas cuestiones que, por otra parte, son muy frecuentes, tenemos las instrucciones de control de flujo de programa. Estas instrucciones de control son compartidas por todos los lenguajes de programación, puede haber ligeras diferencias en los nombres de las instrucciones, pero su semántica es la misma.

#### En C# contamos con varios tipos:

- ❖ Las instrucciones de decisión (if...else, switch) son aquellas que permiten ejecutar bloques de instrucciones sólo si se cumple una determinada condición, o si no se cumple esa condición.
- ❖ Las instrucciones cíclicas o iterativas (while, do while, for, foreach) son aquellas que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces.
- ❖ Las instrucciones de salto (break, continue, return, goto) permiten variar el orden normal en que se ejecutan las instrucciones de un programa, que consiste en ejecutarlas una tras otra en el mismo orden en que se hubiesen escrito en el código fuente. Si bien se mencionan, no es buena práctica usar las instrucciones de salto continue y goto.

## Herramienta de Decisión Simple - IF - ELSE - ELSE IF

La herramienta de decisión simple es el If, su traducción del inglés sería “si” y nos sirve para realizar comparaciones lógicas, por ejemplo, si queremos saber si un alumno aprobó la materia, y tenemos el dato fijo de que la materia se aprueba con 7 o más, podemos realizar la consulta comparando la calificación del alumno contra el dato fijo a través con la utilización del if. A continuación, colocare la comparativa de este ejemplo en pseudocódigo y código C#:

PSEUDOCODIGO	C#
<ul style="list-style-type: none"> <li>• Creo una variable decimal que contendrá la calificación.</li> <li>• Pido al usuario que ingrese la nota</li> <li>• Leo la nota que ingreso el usuario</li> <li>• Si la nota es mayor o igual a 7</li> <li>• Informo que el alumno aprobó.</li> <li>• fin</li> </ul>	<pre>double nota; Console.WriteLine("Ingreso Nota:"); nota = Convert.ToDouble(Console.ReadLine()); if (nota &gt;= 7) {     Console.WriteLine("APROBADO"); } Console.ReadKey();</pre>

Ahora bien, supongamos que además de informar si aprobó, debo informar que esta desaprobado, aquí es donde agregamos la utilización del **else** (si no), veámoslo en otro ejemplo:

PSEUDOCODIGO	C#
<ul style="list-style-type: none"> <li>• Creo una variable decimal que contendrá la calificación.</li> <li>• Pido al usuario que ingrese la nota</li> <li>• Leo la nota que ingreso el usuario</li> <li>• Si la nota es mayor o igual a 7</li> <li>• Informo que el alumno aprobó.</li> <li>• Si no</li> <li>• Informo que el alumno desaprobó</li> <li>• fin</li> </ul>	<pre>double nota; Console.WriteLine("Ingrese Nota:"); nota = Convert.ToDouble(Console.ReadLine()); if (nota &gt;= 7) {     Console.WriteLine("APROBADO"); } else {     Console.WriteLine("DESAPROBADO"); } Console.ReadKey();</pre>

Pero ahora, nos dicen que además de lo anterior, también debemos informar que, si la calificación del alumno es menor que 7 y mayor o igual a 4, el alumno deberá rendir examen en diciembre, y si es menor que 4 rendirá examen en febrero. Lo veremos en el siguiente ejemplo:

PSEUDOCODIGO	C#
<ul style="list-style-type: none"> <li>• Creo una variable decimal que contendrá la calificación.</li> <li>• Pido al usuario que ingrese la nota</li> <li>• Leo la nota que ingreso el usuario</li> <li>• Si la nota es mayor o igual a 7</li> <li>• Informo que el alumno aprobó.</li> <li>• Si no, si la nota es menor que 7 y mayor o igual a 4</li> <li>• Informo que el alumno rinde en Diciembre</li> <li>• Si no</li> <li>• Rinde en Febrero</li> <li>• fin</li> </ul>	<pre>double nota; Console.WriteLine("Ingrese Nota:"); nota = Convert.ToDouble(Console.ReadLine()); if (nota &gt;= 7) {     Console.WriteLine("APROBADO"); } else if (nota &gt;= 4) {     Console.WriteLine("DICIEMBRE"); } else {     Console.WriteLine("FEBRERO"); } Console.ReadKey();</pre>

Presten atención, que en el caso del código en el Else If, solo comparo que sea mayor o igual a 4 para diciembre, en este caso, no es necesario que compare nuevamente a que sea menor que 7, dado que el programa ya comparó anteriormente si la nota era mayor que 7, y dicho resultado dio falso.

Cabe mencionar que, toda comparación lógica, dará un resultado true o false (se cumple o no se cumple), y podemos combinar varias comparaciones lógicas, al igual que en las “tablas de verdad”.

## ***Herramienta de Decisión Múltiple (SWITCH)***

### **Uso de Switch**

El significado de esta instrucción es el siguiente: se evalúa la condición y se ejecutará el código cuando coincida el valor con cada caso. Si no es igual a ninguno de esos valores y se incluye la rama default; pero si no se incluye se pasa directamente a ejecutar la instrucción siguiente al switch.

Los valores indicados en cada rama del switch han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo char o de tipo string. Además, no puede haber más de una rama con el mismo valor.

En realidad, aunque todas las ramas de un switch son opcionales siempre se ha de incluir al menos una. Además, la rama default es opcional, pero es recomendable que se incluya para facilitar la legibilidad del código.

El elemento marcado como colocado tras cada bloque de instrucciones indica qué es lo que ha de hacerse tras ejecutar las instrucciones del bloque que lo preceden.

Puede ser uno de estos tres tipos de instrucciones:

- ❖ `case <valor>;`
- ❖ `default;`
- ❖ `break;`

Si es un case indica que se ha de seguir ejecutando el bloque de instrucciones asociado en el switch a la rama del <valor> indicado.

```
using System;

namespace UsoSwitch
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        // aca pueden cambiar el valor de la/las variables
        // para ver los diferentes resultados posibles.
        bool Booleana = true;
        char Caracter = 'a';
        string cadena = "Mañana"; // Tarde // noche
        int Nro = 7;

        // Con variables booleanas (bool)
        switch (Booleana)
        {
            case true :
                Console.WriteLine("La variable entro como Verdadero");
                Console.ReadKey();
                break;
            case false :
                Console.WriteLine("La variable entro como Falso");
                Console.ReadKey();
                break;
        }

        //Con variable caracter (char)
        switch (Caracter)
        {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                Console.WriteLine("El caracter es vocal");
                Console.ReadKey();
                break;
            default:
                Console.WriteLine("El caracter es consonante");
                Console.ReadKey();
                break;
        }

        // Con variable de cadenas (string)
        switch (cadena)

```

```

{
    case "Mañana":
        Console.WriteLine("que tengas un buen dia");
        Console.ReadKey();
        break;
    case "Tarde":
        Console.WriteLine("que tengas una buena tarde");
        Console.ReadKey();
        break;
    case "Noche":
        Console.WriteLine("que tengas buenas noches");
        Console.ReadKey();
        break;
}

//Con variables numericas (int)
switch (Nro)
{
    case 3:
        //.....
        break;
    case 4:
        //.....
        break;
    default:
        //.....
        break;
}
}
}
}

```

Ahora analicemos el código, lo primero son 4 variables de diferentes tipos, con las cuales armaremos los diversos switches más comunes que podemos llegar a trabajar.

¿Qué hace el Switch? Para explicarlo en pocas palabras, dependiendo del valor de la variable con la que trabajamos, realiza diferentes tareas. Podríamos decir que, en algunos casos, reemplazaría a los If/Elseif. Su estructura arranca con la palabra Switch seguida de la variable a evaluar encerrada entre paréntesis; inmediatamente, como casi todo en C#, abrimos y cerramos llaves, dentro de esa estructura irán los diferentes Case (casos), seguido del Case, espacio de por medio, viene el valor de la variable con el cual queremos que el programa realice determinada tarea, luego de la/las instrucción/es, viene el Break, esto hará que el sistema

se detenga a realizarlas, si obviáramos colocar el Break el sistema continuará hasta el siguiente break que cumpla la condición, si observan bien en el switch del Char con las vocales y las consonantes, ahí utilizo esta opción (nos sirve por ej. para hacer la misma tarea para un rango de valores).

## *Ciclos de repetición*

### **For**

La instrucción “for” ejecuta un ciclo determinado, un número de veces. Consta de tres partes:

- La sentencia inicialización, que se ejecuta al comienzo del ciclo.
- La condición, que se evalúa al entrar al ciclo y luego al comienzo de cada iteración.
- La modificación, que se ejecuta al final de cada iteración, y previene que no se entre un ciclo infinito.

Su sintaxis es:

```
for(<inicialización>; <condición>; <modificación>)  
{  
    <instrucciones>  
}
```

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de <inicialización>, que suelen usarse para definir e inicializar variables que luego se usarán en <instrucciones> (sentencias). Luego se evalúa <condición>, y si es falsa se continúa ejecutando por la instrucción siguiente al for; mientras que si es cierta se ejecutan las <instrucciones> indicadas, luego se ejecutan las instrucciones de <modificación> - que como su nombre indica suelen usarse para modificar los valores de variables que se usen en <instrucciones> - y luego se reevalúa <condición> repitiéndose el proceso hasta que ésta última deje de ser verdadera.

En <inicialización> puede, en realidad, incluirse cualquier número de instrucciones que no tienen por qué ser relativas a inicializar variables o modificarlas, aunque lo anterior sea su uso más habitual. En caso de ser varias se han de separar mediante comas (,), ya que el carácter de punto y coma (;) habitualmente se usa en el for para separar los bloques de <inicialización>, <condición> y <modificación>. Además, la instrucción nula no se puede usar en este caso y tampoco pueden combinarse definiciones de variables con instrucciones de otros tipos.

Con <modificación> pasa algo similar, ya que puede incluirse código que nada tenga que ver con modificaciones, pero en este caso no se pueden incluir definiciones de variables.

Además, las variables que se definan en <inicialización> serán visibles sólo dentro de las <instrucciones>. Al igual que con while, dentro de las <instrucciones> del for también pueden incluirse instrucciones continue; y break; que puedan alterar el funcionamiento normal del bucle.

**Ejemplo:**

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CicloFor
{
    class Program
    {
        static void Main(string[] args)
        {
            /*
             * Si observamos el código a continuación, veremos que el
             * for tiene 3 secciones encerradas entre paréntesis y separadas
             * por un (;), la 1ra sección indica en que valor comenzara
             * el ciclo, la 2da indica hasta que valor se repetirá, y la 3ra
             * es el incremento.
             */

            int i = 0;

            Console.WriteLine("Ciclo For clasico");
            Console.WriteLine("1er ciclo for");
            for (i = 0; i < 5; i++)
            {
                Console.WriteLine(i);
            }
            Console.WriteLine("");
            /*
             * en esta línea, vemos en que valor quedara la variable
             * incremental, y podremos observar que siempre quedara
             * en el valor final más 1.
             */
            Console.WriteLine("Valor en que queda la variable fuera del ciclo: {0}",
i);

            Console.WriteLine("");

            /*
             * Tambien existe la posibilidad en estos ciclos de obviar
             * la primera seccion, en el ej siguiente vemos que es lo que sucede

```



```

    * y al ejecutar el programa veremos que la variable incremental
    * comienza donde termino el ciclo anterior (para obviar la 1er seccion
    * debemos mantener el (;) separador).
    */
    Console.WriteLine("2do ciclo for iniciando la variable en su valor previo
(obviando el parámetro de inicio).");
    for ( ; i < 10; i++)
    {
        Console.WriteLine(i);
    }
    Console.WriteLine("");

    /*
    * En el siguiente ciclo veremos que tambien existe la posibilidad de
obviar

    * la 3ra seccion, pero en esete caso, deberemos tener mucho cuidado
    * dado que SI DENTRO DEL CICLO NO INCREMENTAMOS LA VARIABLE, ESTE ENTRARIA
    * EN UN BUCLE INFINITO.
    */
    Console.WriteLine("3er ciclo For, colocando su incremento dentro de la
estructura de instrucciones.");
    for (i = 0; i < 5; )
    {
        Console.WriteLine(i);
        i++; // Aqui incremento la variable.
    }
    Console.WriteLine("");

    /*
    * En el proximo ciclo veremos que podemos utilizar mas de una variable
    * incremental o decremental
    */
    Console.WriteLine("Ciclo for con 2 variables incrementales y/o
decrementales.");
    int m = 5;
    for (i = 0, m = 5; i<5; i++ , m--)
    {
        Console.WriteLine("Valor de i es: {0}, el valor de m es: {1}", i , m);
    }
    Console.WriteLine("");

    /*

```

```

        * Tambien podemos incrementar la variable de control
        * en 2, 3, o lo que necesitemos
        */
        Console.WriteLine("Ciclo for con incrementando la variable de control
2.");

        for (i = 0; i < 5; i+=2)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine("");

        // De la misma manera que en el ciclo anterior, podemos incrementarla
        exponencialmente.
        Console.WriteLine("Ciclo for incrementando la variable de control
exponencialmente.");
        for (i = 2; i <= 10000; i*=i)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine("");

        Console.ReadKey();
    }
}

```

Vale aclarar que en la segunda sección del For, donde colocamos hasta donde queremos que se incremente o decremente el ciclo, también podemos colocar una llamada a un método que devuelva true o false, de esta manera el ciclo comprobará a cada paso que está devolviendo dicho método, y dependiendo de ello continuar o salir del ciclo.

Otra manera en la que podemos utilizar nuestros ciclos For, es en lugar de utilizar variables incrementales enteras, es que podemos utilizar variables de texto, a continuación, les coloco un ejemplo:

```

Console.WriteLine("Ciclo for utilizando la variable de control de tipo Char.");
char letra ;
for (letra = 'a'; letra <= 'z'; letra++)
{
    Console.WriteLine(letra);
}

```

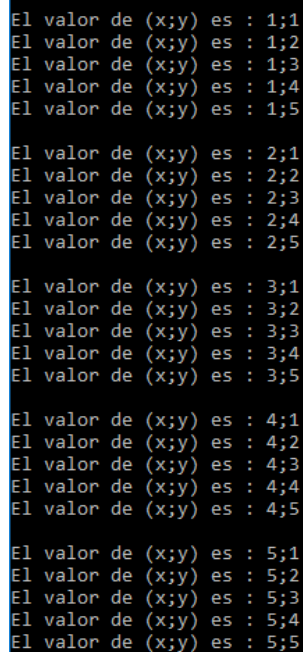
## For Anidados

Los for anidados son como los if anidados, consiste en colocar un ciclo for dentro de otro, esto hará que por cada vez que se ejecute el ciclo externo, se ejecutaran todas las "vueltas" del ciclo interno.

A continuación, coloco un ejemplo:

```
int x;
int y;
for (x=1; x<= 5; x++)
{
    for (y=1; y<= 5; y++ )
    {
        Console.WriteLine("El valor de (x;y) es : {0};{1}", x, y);
    }
    Console.WriteLine();
}
Console.ReadKey();
```

El código daría el siguiente resultado:



```
El valor de (x;y) es : 1;1
El valor de (x;y) es : 1;2
El valor de (x;y) es : 1;3
El valor de (x;y) es : 1;4
El valor de (x;y) es : 1;5
El valor de (x;y) es : 2;1
El valor de (x;y) es : 2;2
El valor de (x;y) es : 2;3
El valor de (x;y) es : 2;4
El valor de (x;y) es : 2;5
El valor de (x;y) es : 3;1
El valor de (x;y) es : 3;2
El valor de (x;y) es : 3;3
El valor de (x;y) es : 3;4
El valor de (x;y) es : 3;5
El valor de (x;y) es : 4;1
El valor de (x;y) es : 4;2
El valor de (x;y) es : 4;3
El valor de (x;y) es : 4;4
El valor de (x;y) es : 4;5
El valor de (x;y) es : 5;1
El valor de (x;y) es : 5;2
El valor de (x;y) es : 5;3
El valor de (x;y) es : 5;4
El valor de (x;y) es : 5;5
```

El ejemplo más claro sería para recorrer todas las posiciones de una matriz.

## Foreach

El Foreach es un ciclo especial, muy útil al momento de recorrer, por ejemplo, los objetos de un formulario, de un contenedor, los caracteres de un string, una matriz, etc. Su estructura interna es similar al ciclo For, a continuación, se muestran algunos ejemplos:

Este ejemplo recorre los caracteres de un string y los muestra uno en cada línea.

```
namespace CicloForeach
{
    class Program
    {
        static void Main(string[] args)
        {
            string apellido = "Ferrando";
            foreach (char letra in nombre)
            {
                Console.WriteLine(letra);
            }
            Console.ReadKey();
        }
    }
}
```

Lo que se realiza en este ejemplo radica en definir una variable de ámbito de bloque llamada “letra”, que va a ser de tipo “char”; y va a recorrer una cadena de caracteres denominada “nombre”. Cada vez que entra a la cadena “nombre” va a encontrar una nueva letra y la va a guardar en la variable creada anteriormente. De esta manera, podremos recorrer toda la cadena y realizar las acciones necesarias para cada carácter.

El siguiente ejemplo son 2 métodos de una clase que reciben por parámetro un formulario, y recorren todos los controles del mismo y dependiendo del control cambia su propiedad Enabled a False.

```
public void BloquearObjetos(Form FRM)
{
    foreach (Control c in FRM.Controls)
    {
        if (c is TextBox)
        {
            ((TextBox)c).Enabled = false;
        }
        if (c is ComboBox)
        {
            ((ComboBox)c).Enabled = false;
        }
        if (c is GroupBox | c is Panel)
        {
            LC2(c);
        }
    }
}
```

```

private static void LC2(Control x)
{
    foreach (Control h in x.Controls)
    {
        if (h is TextBox)
        {
            ((TextBox)h).Enabled = false;
        }
        if (h is ComboBox)
        {
            ((ComboBox)h).Enabled=false;
        }
    }
}

```

## **Bucles**

Los bucles While y Do While, son ciclos de repetición que se ejecutarán mientras se cumpla una determinada condición, al momento de no cumplirse dicha condición el bucle terminara. La diferencia entre ambos bucles radica en el lugar donde ira la condición de salida, para el caso del While, la condición ira al principio del bucle, mientras que, para el otro, ira al final, garantizándose así la ejecución del código interno del bucle por lo menos una vez.

También existen lo que se denomina “Bucles infinitos” estos bucles se ejecutan indefinidamente, pero estos bucles se utilizan para casos muy particulares.

Imaginemos que, por algún motivo, necesitemos que nuestro bucle termine en alguna parte especifica de su código antes que termine el mismo (por ej. a la mitad, si es que se cumple una condición muy particular), para ello tenemos la instrucción Break, esta instrucción detiene el bucle y el programa continua con la instrucción inmediatamente posterior al mismo.

## ***While***

La instrucción while permite ejecutar un bloque de instrucciones mientras la condición sea verdadera. Su sintaxis de uso es:

```

while(<condición>)
{
    <instrucciones>
}

```

Se evalúa la indicada, que ha de producir un valor lógico. Si la condición es cierta (es decir, true) se ejecutan las <instrucciones> (sentencia o sentencias dentro del ciclo) y se repite el proceso de evaluación de <condición> y la ejecución de <instrucciones> hasta que deje de ser true.

Cuando la condición sea falsa (false) se pasará a ejecutar la instrucción siguiente al while. En realidad, <instrucciones> puede ser una única sentencia o un bloque de sentencias.

Por otro lado, dentro de las <instrucciones> de un while pueden utilizarse las siguientes dos instrucciones especiales (las mencionamos como parte de las instrucciones de programación básicas, pero no recomendamos su uso, recomendamos armar expresiones relacionales y/o lógicas correctas y robustas).

- ❖ **break;** Indica que se ha de abortar la ejecución del ciclo y continuar la ejecución en la instrucción siguiente al while.
- ❖ **continue;** Indica que se debe abortar la ejecución de las <instrucciones> y reevaluar la <condición> del ciclo. Si la <condición> resultara cierta, se vuelven a ejecutar las <instrucciones>, en cambio, si es falsa se pasa a ejecutar la instrucción siguiente al while.

```
bool encendido = true;
int i;
int x = 0;
//mientras se cumpla la condición que encendido valga true
while (encendido)
{
    for (i=1; i<=5; i++)
    {
        Console.WriteLine("el valor de la variable es {0}", encendido);
    }
    Console.WriteLine();
    i++;
    if (x == 2) //condición de salida si se cumple determinada acción
    {
        encendido = false;
        Console.WriteLine("el valor de la variable es {0}", encendido);
    }
    Console.WriteLine("el valor de la variable x es {0}", x);
    x++;
}
Console.ReadKey();
```

## Do / While

La instrucción do...while es una variante del while que se utiliza de la siguiente manera:

```
do
{
    <instrucciones>
}
while(<condición>);
```

La única diferencia del significado de do...while respecto al de while es que en vez de evaluar primero la condición y ejecutar <instrucciones> (sentencia) sólo si es verdadera, do...while primero ejecuta las <instrucciones> y luego verifica la <condición> para ver si se ha de repetir la ejecución de las mismas. Por lo demás ambas instrucciones son iguales, e incluso también puede incluirse break; y continue; entre las <instrucciones> del do...while.

do ... while está especialmente destinado para los casos en los que haya que ejecutar las <instrucciones> al menos una vez, aun cuando la condición sea falsa desde el principio.

Ejemplo:

```
do
{
    for (i=1; i<=5; i++)
    {
        Console.WriteLine("el valor de la variable es {0}", encendido);
    }
    Console.WriteLine();
    i++;
    if (x == 2) //condicion de salida si se cumple determinada accion
    {
        encendido = false;
        Console.WriteLine("el valor de la variable es {0}", encendido);
    }
    Console.WriteLine("el valor de la variable x es {0}", x);
    x++;
} while (encendido);

Console.ReadKey();
```

## **Instrucción Break**

Ya se ha comentado que la instrucción break sólo puede incluirse dentro de bloques de instrucciones asociados a instrucciones iterativas o instrucciones switch e indica que se desea abortar la ejecución de las mismas y seguir ejecutando a partir de la instrucción siguiente a ellas. Se usa así: break;

### **Ejemplo:**

```
static void Break_Ejemplo()
{
    for (int numero = 2; numero <= 30; numero += 2)
    {
        Console.WriteLine(numero);

        if (numero == 20)
        {
            Console.WriteLine("Llegó a 20 y salió con Break.");
            break;
        }
    }
    Console.WriteLine("Aquí salió con el break");
}
```

Cuando esta sentencia se usa dentro de un bloque try con cláusula finally (que tiene como objetivo atrapar errores en tiempo de ejecución), antes de abortarse la ejecución de la instrucción iterativa o del switch que la contiene y seguirse ejecutando por la instrucción que le siga, se ejecutarán las instrucciones de la cláusula finally del try. Esto se hace para asegurar que el bloque finally se ejecute aún en caso de salto. Además, si dentro una cláusula finally incluida en un switch o de una instrucción iterativa se usa break, no se permite que como resultado del break se salga del finally.

## **Instrucción Continue**

Ya se ha comentado que la instrucción continue sólo puede usarse dentro del bloque de instrucciones de una instrucción iterativa e indica que se desea pasar a reevaluar directamente la condición de la misma sin ejecutar el resto de instrucciones que contuviese. La evaluación de la condición se haría de la forma habitual: si es cierta se repite el ciclo y si es falsa se continúa ejecutando por la instrucción que le sigue. Su sintaxis de uso es así de sencilla: **continue;**



**Ejemplo:**

```
static void Continue_Ejemplo()
{
    for (int numero = 2; numero <= 30; numero += 2)
    {
        Console.WriteLine(numero);

        if (numero == 20)
        {
            Console.WriteLine("con Continue vuelve a  
ingresar al ciclo. No sale nunca");
            numero = 8;
            continue;
        }
    }
}
```

En cuanto a sus usos dentro de sentencias try, tiene las mismas restricciones que break: antes de salir de un try se ejecutará siempre su bloque finally y no es posible salir de un finally incluido dentro de una instrucción iterativa como consecuencia de un continue.

## **Instrucción Return**

Esta instrucción se usa para indicar cuál es el objeto que ha de devolver una función. Su sintaxis es la siguiente:

**return ;**

La ejecución de esta instrucción provoca que se aborte la ejecución de la función dentro del que aparece y que se devuelva el <objetoRetorno> a la función que lo llamó. Como es lógico, este objeto ha de ser del tipo de retorno de la función en que aparece el return o de alguno compatible con él, por lo que esta instrucción sólo podrá incluirse en funciones cuyo tipo de retorno no sea void, o en los bloques get de las propiedades. De hecho, es obligatorio que toda función con tipo de retorno termine por un return.

Las funciones que devuelvan void pueden tener un return con una sintaxis especial en la que no se indica ningún valor a devolver, sino que simplemente se usa return para indicar que se desea terminar la ejecución del método: **return;**

**Ejemplo:**

```
static int Return_Ejemplo_Funcion()
{
    int numero1 = 10, numero2 = 20;
    return numero1 + numero2;
}
```

```
static void Return_Ejercicio4()
{
    for (int numero = 2; numero <= 30; numero += 2)
    {
        Console.WriteLine(numero);

        if (numero == 20)
        {
            Console.WriteLine("Sale con Return.....");
            return;
        }
    }
}
```

Nuevamente, como con el resto de las instrucciones de salto hasta ahora vistas, si se incluyese un return dentro de un bloque try con cláusula finally, antes de devolverse el objeto especificado se ejecutarían las instrucciones de la cláusula finally. Si hubiese varios bloques finally anidados, las instrucciones de cada uno se ejecutarían de manera ordenada (o sea, del más interno al más externo). Ahora bien, lo que no es posible es incluir un return dentro de una cláusula finally.

**Instrucción Goto**

La instrucción goto permite pasar a ejecutar el código a partir de una instrucción cuya etiqueta se indica en el goto. La sintaxis de uso de esta instrucción es: **goto <etiqueta>;**

Como en la mayoría de los lenguajes, goto es una instrucción nada recomendable cuyo uso dificulta innecesariamente la legibilidad del código y suele ser fácil simularla usando instrucciones iterativas y selectivas con las condiciones apropiadas. Sin embargo, en C# se incluye porque puede ser eficiente usarla si se anidan muchas instrucciones y para reducir sus efectos negativos se le han impuesto unas restricciones:

- ❖ Sólo se pueden etiquetar instrucciones, y no directivas de preprocesado, directivas using o definiciones de miembros, tipos o espacios de nombres.
- ❖ La etiqueta indicada no pueda pertenecer a un bloque de instrucciones anidado dentro del bloque desde el que se usa el goto, ni que etiquete a instrucciones de otro método diferente a aquél en el cual se encuentra el goto que la referencia.
- ❖ Para etiquetar una instrucción con goto basta precederla del nombre con el que se la quiera etiquetar seguido de dos puntos (:). Por ej., el siguiente código demuestra cómo usar goto y definir una etiqueta:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GOTO
{
    class SalirGOTO
    {
        static void Main(string[] args)
        {
            for (int numero = 1; numero <= 5; numero++)
            {
                if (numero <= 3)
                {
                    Console.WriteLine(numero);
                }
                else
                {
                    goto salida;
                }
            }
            salida: Console.WriteLine("Llegó a 3 e interrumpió la
ejecución del for");
            Console.ReadKey();
        }
    }
}

```

El programa de ejemplo lo que hace es mostrar por pantalla los números del 1 al 5, pero una vez mostrado el 3 se aborta la ejecución de la aplicación. Véase además que este ejemplo pone de manifiesto una de las utilidades de la instrucción nula, ya que si no se hubiese escrito tras la etiqueta fin el programa no compilaría en tanto que toda etiqueta ha de preceder a alguna instrucción (aunque sea la instrucción nula) Nótese que al fin y al cabo los usos de goto dentro de instrucciones switch que se vieron al estudiar dicha instrucción no son más que variantes del uso general de goto, ya que default: no es más que una etiqueta y case <valor>: puede verse como una etiqueta un tanto especial cuyo nombre es case seguido de espacios en blanco y un valor. En ambos casos, la etiqueta indicada ha de pertenecer al mismo switch que el goto usado.

## ESTRUCTURAS DE DATOS

### *VECTOR / ARRAY / MATRICES / ARREGLOS*

Podríamos definir a un vector, como una variable dividida en una determinada cantidad de espacios, o como una estructura de datos que permite almacenar un conjunto de datos siempre del mismo tipo, es decir, si necesitamos almacenar 10 números enteros, podríamos crear para ello un vector de 10 posiciones de tipo int. Un Array se lo declara de la siguiente forma: tipo[ ] variable; Es muy parecido a como se declara una variable normal, sólo que hay que poner corchetes detrás del tipo de dato. En C# los arrays son objetos derivados de la clase System.Array. Por lo tanto, y esto es muy importante, cuando declaramos un array en C# este aún no

se habrá creado en la memoria de la computadora, en consecuencia, antes de poder usarlos habrá que instanciarlos, como si fuera cualquier otro objeto.

**int[] Num = new int [10];**

**int** -> Tipo de dato a guardar (entero) – esto depende del tipo de dato que queramos guardar (int – double – float – string ).

**[ ]** -> Indica que es un vector.

**Num** -> Nombre del Vector.

**= New int [10]** -> Inicializa el vector con 10 posiciones.

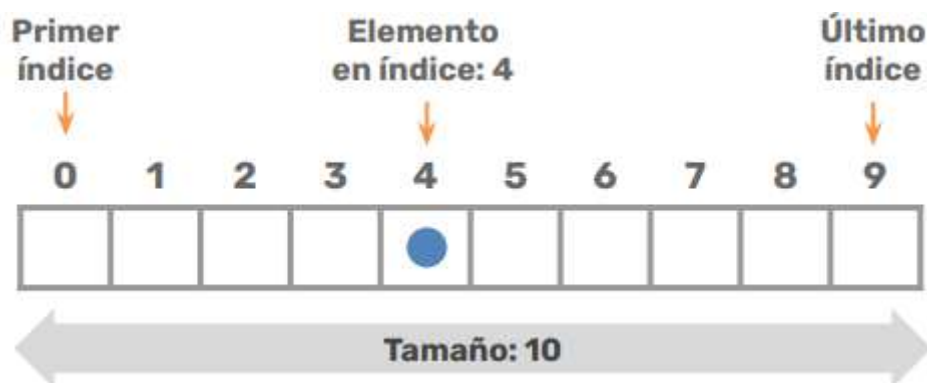
Cabe aclarar que tanto la asignación como la instanciación pueden encontrarse separadas de la siguiente manera:

**int[] Num;** -> Asignación.

**Num = New int[10];** -> Instanciación.

(Esto se debe a que la instanciación reserve los espacios en memoria que utilizara el vector).

Si lo graficáramos quedaría algo así:



Cada componente del vector, se encuentra referenciado por un subíndice, por el cual haremos referencia a dicho componente, cabe aclarar, que los índices comienzan por 0 (cero) y **NO** por 1 (uno). De esta manera, si quisiéramos hacer referencia al dato guardado, por ejemplo, en la posición 3 del vector, solo deberíamos colocar el nombre del vector y entre los corchetes colocar la posición.

```
int[] Num = New int[5];
Num[0]=5;
Num[1]=7;
Num[2]=2;
Num[3]=10;
Num[4]=8;
```

5	7	2	10	8
---	---	---	----	---

Un array es un conjunto ordenado de objetos de tamaño fijo. Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo [] sobre la tabla para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder dentro del conjunto. Es decir, este operador se usa así: [<posicionElemento>]

`Int x = Num[3];` --> La variable x obtendría el valor 10.

O

`Console.WriteLine(Num[3]);` --> Imprimiría por consola el valor 10.

**Para asignarle un valor:**

`Num[3] = x;` --> Si cargamos desde una variable.

O

`Num[3] = Convert.ToInt32(Console.ReadLine());` --> Si cargamos desde el ingreso por consola.

Otra manera en la que podemos cargar un vector es al momento de instanciarlo si ya sabemos los datos:

`String[] DiasSemana = new string[] { "Lu", "Ma", "Mi", "Ju", "Vi", "Sa", "Do"};`

O

`int[] arr = new int[] { 1, 9, 6, 7, 5, 9 };`

## Recorrer Vectores

Bien, ¿ahora como recorreremos un vector entero? para esto utilizamos, de manera óptima, la estructura de repetición foreach. La manera de hacerlo sería la siguiente:

En el foreach se declara una variable del mismo tipo del objeto que se va a recorrer, en este caso un vector, luego sigue la instrucción "in" y el nombre del objeto, esto significaría que el foreach va a recorrer ese vector del principio hasta el final, y que, en cada paso, cargara el dato que se encuentre en cada componente (subíndice) en la variable. Los ejemplos siguientes muestran como recorrer el vector e imprimir por consola.

### Ejemplo 1

```
foreach (string variable in DiasSemana )
{
    Console.WriteLine(variable + " ");
}
Console.ReadKey();
```

### Ejemplo 2

```
foreach (int variable in arr)
```

```
{
    Console.WriteLine(variable + " ");
}
```

**Ejemplo 3**

```
for (int i=0 ; i < NombreArray.Length ; i++)
{
    Console.WriteLine(NombreArray[i]);
}
```

**Array Implícito**

Un array implícito es un array que almacena datos, pero no especificamos ni el tipo de datos ni cuantos elementos tendrá.

**Ejemplo:**

```
Var valores = new [] {"elementos"};
```

**Propiedad Length**

Esta propiedad nos da como resultado la cantidad de posiciones del vector.

Console.WriteLine(DiasSemana.Length); Imprime en consola 7.

**La Clase Array**

La clase Array que nos provee C#, es una “clase base”, es decir, predefinida. Se trata de un tipo abstracto que provee métodos estáticos y de instancia para realizar operaciones sobre arreglos tales como creación, manipulación, búsqueda y ordenamiento.

Entre ellos encontramos los que a continuación se detallan:

**Array.Sort**

Ordena los elementos en una matriz unidimensional.

**Sintaxis:** Array.Sort(array);

**Array.Reverse**

Invierte los elementos en una matriz unidimensional.

**Sintaxis:** Array.Reverse(array);

## Array.BinarySearch

Busca el índice de un elemento especificado.

**Sintaxis:** Array.BinarySearch(array, "Elemento\_a\_Buscar");

**Ejemplo:**

```
internal class Program
{
    static void Main(string[] args)
    {
        string[] dinosaurs = {"Pachycephalosaurus",
                               "Amargasaurus",
                               "Tyrannosaurus",
                               "Mamenchisaurus",
                               "Deinonychus",
                               "Edmontosaurus"};

        int index = Array.BinarySearch(dinosaurs, "Tyrannosaurus");
        Console.WriteLine("Indice del elemento encontrado: {0}.", index);
        Console.ReadKey();
    }
}
```

SALIDA: 2

## Array.Find

Busca contenido dentro de un elemento de un array.

**Sintaxis:** Array.Find(array, n => n.Contains("Texto\_a\_Buscar"));

**Ejemplo:**

```
static void Main()
{
    string[] tresMaestros = { "Balzac", "Dickens", "Dostoievski"};
    string encontrado = Array.Find(tresMaestros, n => n.Contains("o"));
    Console.WriteLine(encontrado); // Dostoievski
}
```

## GetLength

El método Array.GetLength(Int32) se utiliza para encontrar el número total de elementos presentes en la dimensión especificada de la matriz. Cuando se usa el método GetLength en un array, se debe pasar un

número entero (int32) que representa la dimensión del array de la que deseas obtener la longitud. Por ejemplo, si tienes un array multidimensional, como una matriz de dos dimensiones, puedes pasar 0 o 1 como parámetro para obtener la longitud de la primera o segunda dimensión, respectivamente. Por lo tanto, `array.GetLength(0)` te dará la longitud de la primera dimensión y `array.GetLength(1)` te dará la longitud de la segunda dimensión.

**Sintaxis:** `array.GetLength(int32);`

#### Ejemplo:

```
int[] myarray = {445, 44, 66, 6666667, 78, 878, 1};
int result = myarray.GetLength(0);
Console.WriteLine("Elementos Totales: {0}", result);
```

SALIDA:

Elementos Totales: 7

## Array.Resize

Redimensiona un array.

**Sintaxis:** `Array.Resize(ref array, nueva_cantidad_de_posiciones);`

#### Ejemplo:

```
static void Main(string[] args)
{
    var myArr = new int[5] { 1, 2, 3, 4, 5 };
    Array.Resize(ref myArr, 10);
    for (int i = 0; i < myArr.Length; i++)
    {
        Console.WriteLine("[{0}] : {1}", i, myArr[i]);
    }
    Console.ReadLine();
}
```

## Rank

Obtiene el rango (número de dimensiones) de un array. Por ejemplo, una matriz unidimensional devuelve 1, una matriz bidimensional devuelve 2, y así sucesivamente.

**Sintaxis:** `int dimensiones = MiArray.Rank;`



## Array.ForEach

Este método nos permite ejecutar una función en cada uno de los elementos de un array.

**Sintaxis:** Array.ForEach(array, función);

El método se debe llamar a través de la clase y, como argumentos, primero pasaremos el array a analizar y luego la función que ejecutaremos para cada posición del mismo.

### Ejemplo:

```
using System;
public class Program
{
    static int Main()
    {
        string[] nombres={"juan","pedro","marcela","alejandra"};
        Array.ForEach(nombres, agrandar);
        return 0;
    }
    static void agrandar(string n)
    {
        Console.WriteLine(n.ToUpper());
    }
}
```

Primero definimos un array con una serie de nombres, luego usamos el método ForEach donde pasamos el array anterior y la función, esta función está definida a continuación y en este caso tomara el valor recibido del array, este es enviado automáticamente por el método, y mostramos el resultado de aplicar a ToUpper en cada elemento del array.

## Array.Clear

Este método nos permite vaciar un array ya sea totalmente o algunas de sus posiciones.

**Sintaxis:** Array.Clear(array, inicio, longitud);

Se utiliza la clase seguida del método, el primer argumento es el array en cuestión, luego, pasamos desde que posición del array comenzaremos a limpiar los valores, y la cantidad de posiciones a limpiar.

### Ejemplo:

```
using System;
public class Program
{
```

```

static int Main()
{
    int[] arreglo={22,10,19,76,20,23,1,99};
    for(int i=0; i < arreglo.Length; i++)
        Console.WriteLine(arreglo[i]);
    Array.Clear(arreglo,1,3);
    Console.WriteLine("Despues de limpiado el array");
    for(int i=0; i < arreglo.Length; i++)
        Console.WriteLine(arreglo[i]);
    Array.Clear(arreglo,0,arreglo.Length);
    Console.WriteLine("Limpieza total del array");
    for(int i=0; i < arreglo.Length; i++)
        Console.WriteLine(arreglo[i]);
    return 0;
}
}

```

Primero definimos un array de tipo int con algunos valores, lo siguiente es un bucle for donde mostramos todos los valores de nuestro array, después aplicamos el método a nuestro array para que limpie tres elementos desde la segunda posición del array, mostramos un mensaje indicando esto para luego mostrar nuevamente el array, por ultimo aplicamos el método desde el inicio hasta el final del mismo, indicamos que hacemos una limpieza total del array y por ultimo utilizamos un bucle for para mostrar el nuevo resultado.

Salida	Después de limpiado el array	Limpieza total del array
22	22	0
10	0	0
19	0	0
76	0	0
20	20	0
23	23	0
1	1	0
99	99	0

Observación: Tener en cuenta que no “Elimina” la posición, sino que la vacía.

## **Matrices o vectores multidimensionales**

Las matrices o Vectores multidimensionales, son como los vectores unidimensionales (Array) pero su estructura está compuesta de 2 o más elementos o subíndices, podríamos graficarlo como una grilla. Su sintaxis sería de la siguiente manera:

```
Tipo[ , ] NombreMatriz = new Tipo[Indice, Indice];
```

### **Ejemplo:**

```
int[ , ] notas = new int[2,2];
```

En el ejemplo anterior creamos una matriz de dos dimensiones, podríamos graficarla como 2 filas y 2 columnas.

Posición 0,0	Posición 1,0
Posición 0,1	Posición 1,1

Podemos crear tantas filas y tantas columnas como necesitemos, y para recorrerla utilizaríamos 2 bucles anidados, es decir uno dentro del otro.

## ***Colecciones***

Las colecciones son clases que pertenecen al namespace System.Collections.Generic. Estas colecciones son clases genéricas (como indica el propio namespace) y permiten almacenar elementos. Las colecciones no tienen las limitaciones de los arrays (a cambio de mayor consumo de recursos) y permiten las siguientes acciones:

- Ordenar.
- Añadir elementos en tiempo de ejecución.
- Eliminar.
- Buscar.
- Otros.

## **Colecciones más utilizadas**

Colección	Descripción
List<T>	Parecidos a los array pero con métodos adicionales para agregar, eliminar, ordenar, buscar etc
Queue<T>	Las "colas". Un elemento entra y uno sale. Primero en entrar-primero en salir
Stack<T>	Parecido a las Queue pero con algunas diferencias. Primero en entrar-último en salir
LinkedList<T>	Comportamiento como Queue o Stack pero con acceso aleatorio
HashSet<T>	Listas de valores sin ordenar
Dictionary<Tkey, Tvalue>	Almacena elementos con estructura de clave-valor
SortedList<Tkey, Tvalue>	Igual que los Dictionary pero ordenados

### **List <T>**

**Sintaxis:** List <"TipoDeObjeto"> "nombre" = new List <"TipoDeObjeto"> ();

**Método para agregar elementos:** Add

**Método para eliminar elementos:** RemoveAll, removeAt

**Ejemplo:**

```
internal class Program
{
    static void Main(string[] args)
    {
        List<string> Nombres; //creo una lista de tipo string
        Nombres = new List<string>(); // instancio la lista creada anteriormente

        List<int> Edades = new List<int>(); //creo e instancio la lista en un solo paso

        //DIVERSAS MANERAS DE AGREGAR DATOS A LA LISTA
        Nombres.Add("Juan");
        Nombres.Add("Pedro");
        Nombres.Add("Mariela");
        Nombres.Add("Jose");
        Nombres.Add("Juana");

        Edades.Add(5);
        Edades.Add(6);
        Edades.Add(20);

        /* Inserto un nombre en una posición determina
        * para ello debo indicar el indice de la lista
        * donde necesito intercalarlo
        */
    }
}
```

```

Nombres.Insert(0, "Veronica");

//ELIMINAR ELEMENTOS DE LA LISTA
Nombres.Remove("Pedro"); //puedo remover por el dato que contiene la
                           lista
Nombres.RemoveAt(0);      //o puedo remover por el índice

// Si indico eliminar un valor que no existe C# no indicara ningun error
Nombres.Remove("PEPE");

// Si indico eliminar un indice que no existe C# SI indicara un error
//Nombres.RemoveAt(30);

/* RemoveAll() es el más complejo de los métodos remove,
 * pero definitivamente también el más poderoso.
 * Toma un delegado a un método como parámetro y este método decide
 * si un elemento debe eliminarse o no devolviendo verdadero o falso.
 * Esto le permite aplicar su propia lógica al eliminar elementos
 * y también le permite eliminar más de un elemento a la vez.
 * Los delegados no serán tratados en este momento,
 * debido a que es un tema complejo en esta instancia de la cursada,
 * pero es importante que entiendan lo conveniente que es el método
 * RemoveAll, así que aquí hay un ejemplo:
 */

List<string> ListadDeNombres = new List<string>()
{
    "Jorge",
    "Juan",
    "Javier",
    "Marcelo"
};

ListadDeNombres.RemoveAll(name =>
{
    if (name.StartsWith("J"))
        return true;
    else
        return false;
});
//El ejemplo anterior borra todos los nombres que comienzan con la letra J

//ORDENAR LOS ELEMENTOS DE UNA LISTA
Nombres.Sort(); //ordeno de menor a mayor, o de A a Z
Nombres.Reverse();//invierte el orden de la lista

//CONTAR LOS ELEMENTOS DE UNA LISTA
int cantElementos = Nombres.Count();

//SUMAR LOS ELEMENTOS DE UNA LISTA
int sumaElementos = Edades.Sum();

//PROMEDIAR LOS ELEMENTOS DE UNA LISTA
double promedioElementos = Edades.Average();

//MINIMO LOS ELEMENTOS DE UNA LISTA
double minElementos = Edades.Min();

//MAXIMO LOS ELEMENTOS DE UNA LISTA
double maxElementos = Edades.Max();

//OBTENER EL PRIMER Y EL ULTIMO ELEMENTO DE UNA LISTA

```

```

string primero = Nombres.First();
string ultimo = Nombres.Last();

//VACIAR TODA LA LISTA (ELIMINA TODOS LOS ELEMENTOS)
Edades.Clear();

//RECORRER UNA LISTA

//Con ciclo for
Console.WriteLine("Lista recorrida con ciclo for");
Console.WriteLine("-----");
int i;
for (i=0; i < Nombres.Count; i++)
{
    Console.WriteLine("Elemento {0} = {1}", i, Nombres[i]);
}

Console.WriteLine("");
i = 0;

//Con ciclo Foreach
Console.WriteLine("Lista recorrida con ciclo Foreach");
Console.WriteLine("-----");

foreach (var v in Nombres)
{
    Console.WriteLine("Elemento {0} = {1}", i, v);
    i++;
}

i = 0;
Console.WriteLine("");

//Con expresion Lambda
Console.WriteLine("Lista recorrida con expresion Lambda");
Console.WriteLine("-----");

Nombres.ForEach((v) => Console.WriteLine("Elemento = {0}", v));

Console.WriteLine("");

//LISTAS TIPO <<t>> (OBJETOS Y/O CLASES)
Console.WriteLine("Lista TIPO t (OBJETOS)");
Console.WriteLine("-----");

List<Usuarios> ListaDeUsuarios = new List<Usuarios>()
{
    new Usuarios() { Nombre = "Juan Perez", Edad = 6 },
    new Usuarios() { Nombre = "Marcela Santos", Edad = 34 },
    new Usuarios() { Nombre = "Jose Bermudez", Edad = 8 },
};

ListaDeUsuarios.Add(new Usuarios() { Nombre = "Matias Ferrando ", Edad =
    26 });

List<Usuarios> ListaDeUsuariosOrdenada = ListaDeUsuarios.OrderBy(x =>
    x.Edad).ToList();
for (i = 0; i < ListaDeUsuariosOrdenada.Count; i++)
{
    Console.WriteLine(ListaDeUsuariosOrdenada[i].Nombre + " tiene " +
        ListaDeUsuariosOrdenada[i].Edad + " años");
}
Console.ReadKey();

```

```

    }
}

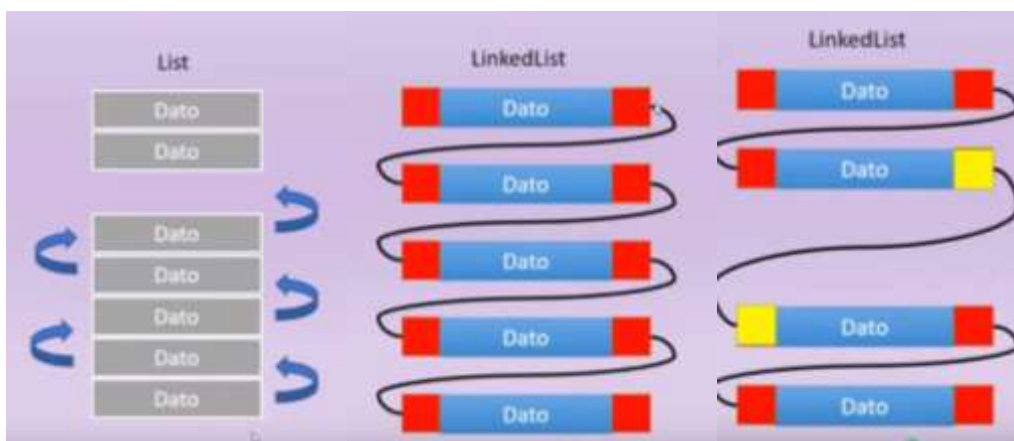
class Usuarios
{
    public string Nombre { get; set; }

    public int Edad { get; set; }
}

```

## LinkedList

La diferencia entre las List y las LinkedList radica en que en las listas los datos **deben** ser adyacentes, por lo que, si se borra un dato, todos los que le siguen deben subir una posición para que no quede ningún espacio vacío. En cambio, las LinkedList tienen dos nodos que indican cual es el dato anterior y el dato posterior, por lo que, si se borra un dato, solo se actualiza el enlace entre los nodos.



### Ejemplo:

```

static void Main(string[] args)
{
    LinkedList<int> numeros = new LinkedList<int>();    // declaramos la linkedlist

    foreach (int numero in new int[] {10,8,6,4,2,0 })
    {
        numeros.AddLast(numero);    // El metodo addfirst agrega los valores en la primera posición, mientras que el addlast agrega al final
    }

    //numeros.Remove(8); //Borra el elemento particular 8

    // agrego un nodo con el valor 15 al principio
    LinkedListNode<int> nodoImportante = new LinkedListNode<int>(15);

    numeros.AddFirst(nodoImportante);

    for (LinkedListNode<int> nodo=numeros.First; nodo != null; nodo = nodo.Next)

```

```

        {
            int numero = nodo.Value;
            Console.WriteLine(numero);
        }
    }
}

```

## **Queue (Listas tipo cola)**

Las listas tipo cola son las listas F.I.F.O. (First In First Out, primero en entrar primero en salir). Se utilizan, por ejemplo, cuando un programa necesita realizar procesos en forma secuencial. Un ejemplo cotidiano sería cuando mandamos varios archivos a imprimir, se van a ir imprimiendo en el orden en que entraron en la impresora (el primero en entrar es el primero en imprimirse). Los métodos más utilizados son Enqueue (agrega elementos) y Dequeue (Elimina Elementos).

### **Ejemplo:**

```

static void Main(string[] args)
{
    Queue<int> numeros = new Queue<int>();

    //rellenado o agregando elementos a la cola
    foreach (int numero in new int[5] { 2,4,6,8,10})
    {
        numeros.Enqueue(numero);
    }

    // recorriendo la cola
    Console.WriteLine("recorriendo queue");

    foreach (int numero in numeros)
    {
        Console.WriteLine(numero);
    }

    Console.WriteLine("eliminando elementos");
    numeros.Dequeue();

    foreach (int numero in numeros)
    {
        Console.WriteLine(numero);
    }
}

```

## **Stack (Listas tipo Pila)**

Las listas tipo pila son las listas L.I.F.O. (Last In First Out, último en entrar primero en salir). Estas listas se pueden ejemplificar con una pila de platos, si vamos apilando platos uno encima de otro, cuando vayamos a



buscar uno vamos a tomar el de arriba de la pila, fue el último en entrar y es el primero en salir, lo mismo ocurre con la apilación de páginas en una aplicación. Sus métodos más comunes son Pop (agrega elementos) y Push (Elimina elementos). Este tipo de lista se maneja exactamente igual a las listas tipo cola únicamente cambiando los métodos correspondientes y el tipo de lista.

Otro ejemplo podría ser la navegación en aplicaciones web: Las pilas pueden ser utilizadas para realizar un seguimiento del historial de navegación en una aplicación web, lo que permite a los usuarios retroceder y avanzar entre las páginas visitadas.

En general, las listas tipo Stack se utilizan en situaciones en las que se necesita mantener un orden específico de elementos y se requiere un acceso rápido a los elementos más recientes agregados, siguiendo el principio LIFO.

## ***Diferencias de rendimiento entre List, matrices y Array Class en C#***

Lo más habitual cuando desarrollamos aplicaciones en C# y queremos introducir un número determinado de elementos dentro de la colección, es tender al uso de una colección de tipo System.Collections.Generic, y concretamente al uso de List<T>.

List<T> nos proporciona una flexibilidad bastante grande y simplifica mucho el trabajo a la hora de introducir en una colección elementos.

En algunas ocasiones no tendremos claro el número de elementos que podemos introducir, pero en otras ocasiones, sabremos de antemano, cuantos elementos en conjunto (el número exacto de elementos) introduciremos dentro de la colección.

Desde el punto de vista del programador, la tendencia a usar List<T> no tiene por qué ser la más adecuada, es más, puede que sea la más nociva, sobre todo cuando hablamos de rendimiento.

Si queremos trabajar con un conjunto de elementos, podremos utilizar, por ejemplo:

- List<T>
- array[]
- Array Class

De cara al rendimiento y para ver qué pasa o qué resultados podemos obtener cuando sabemos el número de elementos que tendrá nuestra colección al usar cualquiera de estas alternativas, se verá un ejemplo de código.

En primer lugar, una clase que cargará dentro de un bucle, un amplio conjunto de registros.

Mediremos el momento en el que se lanza el proceso, y el momento en el que se nos indica que el proceso ha terminado, y de ahí extraeremos los tiempos que nos permitirán analizar el comportamiento.

El código de la clase que usaremos para hacer esta medición es el siguiente:

```
using System;
using System.Collections.Generic;

public class ListArrayTest
{
    public void ListInt(int iterations)
    {
        List<int> li = new List<int>();
        for (int i = 0; i < iterations; i++)
        {
            li.Add(i);
        }
    }

    public void ListObject(int iterations)
    {
        List<object> li = new List<object>();
        for (int i = 0; i < iterations; i++)
        {
            li.Add(i);
        }
    }

    public void ArrayInt(int iterations)
    {
        int[] a = new int[iterations];
        for (int i = 0; i < iterations; i++)
        {
            a[i] = i;
        }
    }
}
```

```

public void ArrayObject(int iterations)
{
    object[] a = new object[iterations];

    for (int i = 0; i < iterations; i++)
    {
        a[i] = i;
    }
}

public void ArrayClassInt(int iterations)
{
    Array li = Array.CreateInstance(typeof(int), iterations);
    for (int i = 0; i < iterations; i++)
    {
        li.SetValue(i, i);
    }
}

public void ArrayClassObject(int iterations)
{
    Array li = Array.CreateInstance(typeof(object), iterations);
    for (int i = 0; i < iterations; i++)
    {
        li.SetValue(i, i);
    }
}
}

```

Este ejemplo, hará 6 pruebas:

- List<int>
- List<object>
- Array<int>

- Array<object>
- Array Class int
- Array Class object

El código C# de nuestro ejemplo de aplicación de consola que se encarga de ejecutar y medir los métodos explicados en la clase anterior queda de la siguiente forma:

```
using System;
using System.Diagnostics;
public class Program
{
    public static void Main(string[] args)
    {
        ListArrayTest listArrayTest = new ListArrayTest();
        Stopwatch stopWatch = new Stopwatch();
        int iterations = 10000;

        stopWatch.Start();
        listArrayTest.ListInt(iterations);
        stopWatch.Stop();
        Console.WriteLine("List<int> => " + stopWatch.ElapsedTicks);
        stopWatch.Reset();

        stopWatch.Start();
        listArrayTest.ListObject(iterations);
        stopWatch.Stop();
        Console.WriteLine("List<object> => " + stopWatch.ElapsedTicks);
        stopWatch.Reset();

        stopWatch.Start();
        listArrayTest.ArrayInt(iterations);
        stopWatch.Stop();
        Console.WriteLine("Array<int> => " + stopWatch.ElapsedTicks);
        stopWatch.Reset();

        stopWatch.Start();
```

```

listArrayTest.ArrayObject(iterations);
stopWatch.Stop();
Console.WriteLine("Array<object> => " + stopWatch.ElapsedTicks);
stopWatch.Reset();

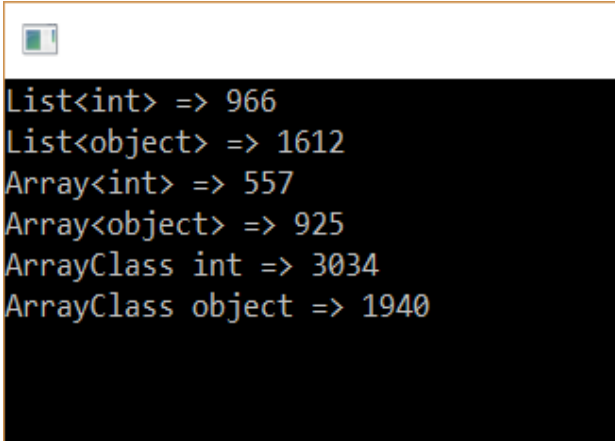
stopWatch.Start();
listArrayTest.ArrayClassInt(iterations);
stopWatch.Stop();
Console.WriteLine("ArrayClass int => " + stopWatch.ElapsedTicks);
stopWatch.Reset();

stopWatch.Start();
listArrayTest.ArrayClassObject(iterations);
stopWatch.Stop();
Console.WriteLine("ArrayClass object => " + stopWatch.ElapsedTicks);
stopWatch.Reset();
Console.ReadKey();
}
}

```

Con 10.000 iteraciones, seremos capaces de tener una idea clara del rendimiento en el uso de una alternativa u otra.

En la siguiente captura, vemos nuestra aplicación en ejecución:



```

List<int> => 966
List<object> => 1612
Array<int> => 557
Array<object> => 925
ArrayClass int => 3034
ArrayClass object => 1940

```

Como podemos apreciar, declarar y añadir elementos a una colección o matriz con un tipo de datos concreto, es más eficiente que hacerlo a una colección o matriz, que con un tipo de datos genérico como lo es object.

De igual forma, declarar y añadir elementos a una matriz de tipo `int[]` es más eficiente que hacerlo a una de tipo `List<int>`, y más aún que a una de tipo `System.Array`.

`List` es peor que `array`, porque almacena el valor como `object`, y luego convierte el tipo valor a un tipo valor por referencia.

`System.Array` por su parte, penaliza aún más esta situación.

En términos generales, una matriz o `array` es recomendable cuando sabemos el número fijo de elementos que va a haber en la colección o el número máximo de elementos posibles. No es recomendable, sin embargo, cuando el número de elementos máximo es desconocido o simplemente no lo sabemos, pudiendo incrementarse o no según la ejecución de los procesos, ya que, al cambiar su dimensión, copiaremos cada elemento en el nuevo `array`.

`List<T>` nos proporciona ventajas a la hora de acceder a los elementos de una colección, y aunque es menos práctica de cara al rendimiento cuando tenemos claro el número máximo de elementos, podría seguir siendo interesante su uso en esas circunstancias si las operaciones que tenemos que hacer con los elementos de la colección son costosas. No obstante, en el caso de desconocer el número máximo o fijo de elementos, es mejor que `array`.

No podemos, por lo tanto, afirmar que algo es mejor que otra cosa. Qué utilizar y cuando, depende de nuestras necesidades.

## MANEJO DE EXCEPCIONES (Control de Errores)

Es muy común en la programación que aparezcan errores inesperados, ya sea por omisión nuestra al codificar o por errores internos o externos. Estos errores suelen cancelar el programa y cerrarlo, incomodando mucho al usuario, y dejando nuestras largas horas de arduo trabajo por el suelo, es por eso que es fundamental controlar estos errores. Vamos a poner un ejemplo muy sencillo, si codificamos un programa que le pida al usuario un número entero, deberemos convertir ese dato ingresado por el usuario que es de formato texto a numérico entero, bien, esto lo realizamos a través de la clase `Convert.ToInt32(...)`, y es más, de omitirlo inteligentemente `C#` nos marca la omisión, hasta acá todo bien, probamos el programa con los datos que esperamos y el programa funciona, y hasta hace los cálculos que esperábamos, podemos suponer que el programa funciona a la perfección, pero, ¿Qué pasaría si el usuario presiona por error un `Enter`, o ingresara una letra? El `Convert.ToInt32(...)` estaría tratando de convertir un valor que no corresponde a un número, con lo que aparece el error, y la aplicación se cierra. Para estos casos (y muchos más) tenemos una herramienta que intercepta estos errores, es un bloque de código denominado `Try/Catch` y su estructura sería la siguiente:

```
try
{
    //código que puede generar el error
    nro = Convert.ToInt32(Console.ReadLine());
}
catch (Exception ex)
{
    string mensaje = "Error : " + ex.Message;
    Console.WriteLine(mensaje);
    // aca pondríamos que queremos que realice el programa en caso de error
}
```

“ex” es una variable que nos ofrece esta herramienta donde guarda información acerca del error, en su método “Message” guarda parte de la información. También podemos decidir no mostrarla, y mostrar un mensaje de error nuestro informando al usuario que debe hacer.

## Métodos – Funciones y Procedimientos

### *Introducción*

Un problema complejo se puede dividir en pequeños subproblemas más sencillos. Estos subproblemas se conocen como módulos y su implementación en un lenguaje de programación se llama subprogramas. Existe una clasificación de subprograma, los procedimientos y las funciones.

Un subprograma realiza las mismas acciones que un programa, sin embargo, vamos a utilizar el subprograma o módulo para una acción u operación específica.

Un subprograma recibe datos de un programa y le devuelve resultados (el programa “llama” o “invoca” al subprograma, éste ejecuta una tarea específica formada por un conjunto de instrucciones y devuelve el “control” al programa que lo llamó).

En C#, a las funciones y procedimientos se los conocen con el nombre de métodos en la programación orientada a objetos (POO).



## Funciones - Retornan un valor

En el ámbito de la programación, una función es un subprograma que contiene una secuencia de instrucciones que realizan una tarea específica dentro de un programa o aplicación más grande.

Las declaraciones de las funciones generalmente son especificadas por:

- ❖ **Un nombre único en el ámbito.** Nombre de la función con el que se identifica y se distingue de otras. No podrá haber otra función o procedimiento con ese nombre (salvo sobrecarga o polimorfismo en programación orientada a objetos (POO)).
- ❖ **Un tipo de dato de retorno.** Tipo de dato del valor que la función devolverá al terminar su ejecución.
- ❖ **Una lista de parámetros.** Especificación del conjunto de parámetros (pueden ser cero, uno o más) que la función debe recibir para realizar su tarea.

### Sintaxis de una función

```

Modificador de acceso Tipo Devuelto Nombre_Funcion (tipo(s) parámetros(s) nombres)
{
    // declaración de datos y cuerpo de la función.
    return (valor);
}
  
```

Ejemplo:

```

static int Funcion_Sumar(int numero1, int numero2)
{
    int resultado = numero1 + numero2;
    return resultado;
}
  
```





## Procedimientos - No retornan valor

Un procedimiento es un subprograma que realiza una tarea específica y es relativamente independiente del resto del código. Los procedimientos suelen utilizarse para reducir la duplicación de códigos en un programa. Los procedimientos pueden recibir parámetros, pero no necesitan devolver un valor como es el caso de las funciones.

### Sintaxis de un procedimiento

```
Modificador de acceso void Nombre_Procedimiento (tipo(s) parámetros(s) nombres)
{
    // declaración de datos y cuerpo del procedimiento.
}
```

### Ejemplo:

```
static void Procedimiento_Sumar(int numero1, int numero2)
{
    int resultado = numero1 + numero2;
}
```

## Parámetros

Un parámetro es un dato que ingresa a una función o procedimiento a través de su llamada. Todos los tipos de C# son tipos de valor o tipos de referencia.

### Pasar parámetros por valor

Cuando un tipo de valor se pasa a un método por valor, se pasa una copia del atributo o variable y no la referencia del propio atributo. Por lo tanto, los cambios realizados en él, en el método llamado, no tienen ningún efecto en el atributo original cuando el control vuelve al autor de la llamada.

En el ejemplo siguiente se pasa un tipo de valor a un método por valor, y el método llamado intenta cambiar el valor del tipo de valor. Define una variable de tipo int, que es un tipo de valor, inicializa su valor en 20 y lo pasa a un método denominado ModificarValor que cambia el valor de la variable a 30. Pero cuando el método vuelve, el valor de la variable no cambia.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Metodos
{
    class MetodoFuncion
    {
        static void Main(string[] args)
        {
            int valor = 20;
            Console.WriteLine("En el Main(), valor tiene = {0}",
valor);
            ModificarValor(valor);
            Console.WriteLine("Volviendo al Main, valor tiene = {0}",
valor);
            Console.ReadKey();
        }
        static void ModificarValor(int i)
        {
            i = 30;
            Console.WriteLine("En ModificarValor(), el valor del
parámetro i es = {0}", i);
            return;
        }
    }
}

```

Y el resultado de su ejecución sería:

```

En el Main(), valor tiene = 20
En ModificarValor(), el valor del parámetro i es = 30
Volviendo al Main, valor tiene = 20

```

Cuando un objeto de un tipo de referencia se pasa a un método por valor, se pasa por valor una referencia al objeto. Es decir, el método no recibe el objeto concreto, sino un argumento que indica la ubicación del objeto, por ende, cabe aclarar, que cualquier objeto (array, una clase, objetos de formularios como combobox, contenedores objetos, etc), aunque lo pasemos por valor, **SIEMPRE** pasará por referencia, aunque no lo especifiquemos. Si cambia un miembro del objeto mediante esta referencia, el cambio se reflejará en el objeto cuando el control vuelva al método de llamada.

En el ejemplo siguiente se define una clase (que es un tipo de referencia) denominada **EjemploRefType**. Crea una instancia de un objeto **EjemploRefType**, asigna 44 a su campo value y pasa el objeto al método **ModificarObjeto**.

Fundamentalmente, este ejemplo hace lo mismo que el ejemplo anterior: pasa un argumento por valor a un método. Pero, debido a que se usa un tipo de referencia, el resultado es diferente. La modificación que se lleva a cabo en **ModificarObjeto** para el campo **obj.value** cambia también el campo **value** del argumento **rt**, en el método **Main** a 33, tal y como muestra el resultado del ejemplo.

#### Ejemplo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Metodos
{
    class MetodoFuncion
    {
        static void Main(string[] args)
        {
            var rt = new EjemploRefType();
            rt.value = 44;
            ModificarObjeto(rt);
            Console.WriteLine(rt.value);
            Console.Read();
        }
        static void ModificarObjeto(EjemploRefType obj)
        {
            obj.value = 33;
        }
    }
    public class EjemploRefType
    {
        public int value;
    }
}
```

### Pasar parámetros por referencia

Pasa un parámetro por referencia cuando quieras cambiar el valor de un argumento en un método y reflejar ese cambio cuando el control vuelva al método de llamada. Para pasar un parámetro por referencia, use la palabra clave **ref**.

El ejemplo siguiente es idéntico al anterior, salvo que el valor se pasa por referencia al método **ModificarValor**. Cuando se modifica el valor del parámetro en el método **ModificarValor**, el cambio del valor se refleja cuando el control vuelve al autor de la llamada.

**Ejemplo:**

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Metodos
{
    class MetodoFuncion
    {
        static void Main(string[] args)
        {
            int valor = 20;
            Console.WriteLine("En el Main(), valor tiene = {0}", valor);
            ModificarValor(ref valor);
            Console.WriteLine("Volviendo al Main, valor tiene = {0}", valor);
            Console.ReadKey();
        }
        static void ModificarValor(ref int i)
        {
            i = 30;
            Console.WriteLine("En ModificarValor(), el valor del parámetro i es = {0}", i);
            return;
        }
    }
}

```

Y el resultado de su ejecución sería:

```

En el Main(), valor tiene = 20
En ModificarValor(), el valor del parámetro i es = 30
Volviendo al Main, valor tiene = 30

```

**Ejemplos**

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Metodos
{
    internal class Program
    {
        static void Main(string[] args)
        {
            /*
             * Un método es un bloque de código que contiene una serie de sentencias.
             * Un programa hace que las declaraciones se ejecuten llamando al método
             * y especificando los argumentos de método requeridos.
             * En C#, cada instrucción ejecutada se realiza en el contexto de un método.
             * -----
             * El método Main es el punto de entrada para cada aplicación de C# y
             * Common Language Runtime (CLR) lo llama cuando se inicia el programa.
             * En una aplicación que utiliza sentencias de nivel superior, el compilador
             * Main genera el método y contiene todas las sentencias de nivel superior.
             * -----
             * Los métodos se declaran en una clase, estructura o interfaz especificando

```

```

* el nivel de acceso como public o private, modificadores opcionales como
* abstract o sealed, el tipo de valor devuelto, el nombre del método y cualquier
* parámetro del método. Estas partes juntas son la firma del método.
* Puede devolver un tipo de dato (bool, double, int, float, string...)
* o no devolver nada (void).
* -----
* Los parámetros del método están entre paréntesis y separados por comas.
* Los paréntesis vacíos indican que el método no requiere parámetros.
* Existen parametros obligatorios (no iniciados con un valor) y optativos
* (iniciados con un valor, estos SIEMPRE van al final), de valor (no modifican
* lo enviado como parametro) o de referencia (modifican el parametro original
enviado)
* -----
*/

Console.WriteLine("Llamo al metodo Mensaje que no recibe ningun parametro:");
Mensaje();
Console.WriteLine("-----");
----" + System.Environment.NewLine);

Console.WriteLine("Llamo al metodo MensajeParametro, que va a recibir un valor
string como parametro");
MensajeParametro("le envio un parametro String");
string mensaje = "valor de una variable String";
MensajeParametro(mensaje);
Console.WriteLine("-----");
----" + System.Environment.NewLine);

// Llamo al metodo TotalProducto para calcular el total de una compra de 5
productos de $250 C/U
Console.WriteLine("Llamo a un metodo que recibe dos parametros obligatorios y
devuelve un valor double:");
Console.WriteLine("El total a pagar es: ${0}", TotalProdcut(5,250));
Console.WriteLine("-----");
----" + System.Environment.NewLine);

// Llamo al metodo TotalProductoOpcionales para calcular el total de una compra
que no paga IVA y no tiene descuentos
Console.WriteLine("Llamo a un metodo que recibe dos parametros obligatorios y dos
optativos, y devuelve un valor double:");
Console.WriteLine("El total a pagar sin IVA ni descuentos es: ${0}",
TotalProductoOpcionales(5,200,pagaIVA:false));
Console.WriteLine("El total a pagar con IVA y descuento del 15% es: ${0}",
TotalProductoOpcionales(5, 200, 15));
Console.WriteLine("El total a pagar sin IVA y con descuento del 15% es: ${0}",
TotalProductoOpcionales(5, 200, 15, false));
Console.WriteLine("-----");
----" + System.Environment.NewLine);

// Hasta este momento todos los metodos que se utilizaron usaron parametros por
valor y no por referencia

Console.WriteLine("Mostramos como se comporta un atributo/variable antes y
despues de enviarlo como parametro por " +
"referencia a un metodo que lo va a incrementar en 5
unidades");
int valor = 20;
Console.WriteLine("El valor antes de llamar al metodo por referencia es: {0}",
valor);
incrementar(ref valor);

```

```

valor);
    Console.WriteLine("-----");
    Console.WriteLine("----" + System.Environment.NewLine);

    /*
     * Aunque no especifiquemos que el metodo incrementarValoresLista va a recibir
     * por referencia, y no le enviemos de
     * esta manera, todos los objetos son referencias, por lo tanto se modificara el
     * original, como es en el caso de
     * las listas, arrays u otros objetos que veremos posteriormente
     */
    List<int> lista = new List<int>();
    lista.Add(5);
    lista.Add(10);
    lista.Add(15);
    Console.WriteLine("Lista original:");
    foreach (var item in lista)
    {
        Console.WriteLine("Valor:{0}", item);
    }
    incrementarValoresLista(lista);

    Console.WriteLine("Lista modificada:");
    foreach (var item in lista)
    {
        Console.WriteLine("Valor:{0}", item);
    }

    Console.ReadKey();
}

/*
 * Declaro un parametro (en VS 2022 al ser en una misma clase no se especifica el
 * nivel de acceso).
 * Este metodo va a ser estatico (debido a que el metodo main que lo llama tambien es
 * estatico) y no
 * va a devolver ningun dato, por lo que se coloca la palabra reservada "void", luego
 * el nombre del
 * metodo y los parentesis vacios ya que no recibe parametros
 */
static void Mensaje()
{
    Console.WriteLine("Esto es un metodo sin parametros que no devuelve ningun
valor");
}

/*
 * Declaro un metodo estatico que no devuelve ningun valor (void), que se llama
 * MensajeParametro y que
 * va a recibir de manera obligatoria un parametro que va a ser String y lo vamos a
 * llamar "mensaje"
 */
static void MensajeParametro(string mensaje)
{
    Console.WriteLine(mensaje);
}

/*
 * Declaro un metodo estatico lue va a calcular el total a pagar de un producto de
 * acuerdo a la cantidad comprada.

```

```

    * Este va a devolver (con la palabra reservada "return") un valor "double",
    * que se va a llamar "TotalProducto" y va a recibir dos parametros obligatorios: la
cantidad de productos (int)
    * y el precio unitario (double)
    */
static double TotalProdcuto (int cantidad, double precio)
{
    double total = cantidad * precio;
    return total;
}

/*
    * Declaro un Metodo estatico con parametros opcionales que va a recibir
obligatoriamente dos valores y optativamente
    * otros dos. Este metodo funcionara como el anterior, pero se le agrega dos opciones
nuevas:
    * indicar el numero de porcentaje de descuento en caso de que exista, y la
posibilidad de especificar que
    * no se le sume el IVA del 21%
    */
static double TotalProductoOpcionales (int cantidad, double precio, int porcDescuento
= 0, bool pagaIVA = true)
{
    double subtotal = cantidad * precio;
    double total = 0;
    // Poniendo solo el nombre de la variable booleana se toma como si la condicion
es que sea verdadera (pagaIVA == true)
    if (pagaIVA)
    {
        total = (subtotal - (porcDescuento * subtotal / 100)) * 1.21;
    }
    else
    {
        total = subtotal - (porcDescuento * subtotal / 100);
    }
    return total;
}

// Creo un metodo estatico que va a recibir la referencia de un parametro al cual le
modificara su valor incrementandolo
static void incrementar (ref int valor)
{
    valor += 5;
}

/*
    * Declaro un metodo que no especifico que va a recibir un valor por referencia, peor
al recibir un objeto, SIEMPRE se
    * recibe una referencia al mismo y se modifica el original
    */
static void incrementarValoresLista (List<int> lista)
{
    for (int i = 0; i < lista.Count; i++)
    {
        lista[i] += 2;
    }
}
}

```

## Clase Math en C#

En C#, la clase Math es una clase estática de la biblioteca de clases .NET que proporciona métodos matemáticos y constantes estándar. Con ella, se pueden realizar operaciones matemáticas como cálculos de raíces cuadradas, logaritmos, potencias, trigonometría y mucho más.

Los métodos y las constantes de Math son comunes a todos los programas de C# y están disponibles sin la necesidad de instanciar un objeto.

Esta clase proporciona métodos para realizar operaciones matemáticas comunes. Algunos de los métodos más comunes son:

- ❖ **Abs:** retorna el valor absoluto de un número.
- ❖ **Cos:** retorna el coseno de un ángulo.
- ❖ **Sin:** retorna el seno de un ángulo.
- ❖ **Tan:** retorna la tangente de un ángulo.
- ❖ **Acos:** retorna el ángulo cuyo coseno es el número especificado.
- ❖ **Asin:** retorna el ángulo cuyo seno es el número especificado.
- ❖ **Atan:** retorna el ángulo cuyo tangente es el número especificado.
- ❖ **Ceiling:** retorna el entero más pequeño que es mayor o igual que el número especificado.
- ❖ **Floor:** retorna el entero más grande que es menor o igual que el número especificado.
- ❖ **Max:** retorna el número más grande de dos números especificados.
- ❖ **Min:** retorna el número más pequeño de dos números especificados.
- ❖ **Pow:** retorna el resultado de elevar un número a una potencia especificada.
- ❖ **Sqrt:** retorna la raíz cuadrada de un número especificado.

## Método Max

El método Max en C# es un método estático (se puede invocar sin tener que crear una instancia de la clase Math) de la clase System.Math que permite determinar el mayor de dos números.

El método Max acepta dos argumentos, y devuelve el mayor de los dos como un resultado. Puede aplicarse a diferentes tipos de números, incluyendo int, double, float y otros tipos numéricos.



**Ejemplo:**

```
int num1 = 10;
int num2 = 20;
int result = Math.Max(num1, num2);
Console.WriteLine("El número mayor es: " + result);
```

En este ejemplo, num1 es igual a 10 y num2 es igual a 20. Al invocar el método Max, se compara num1 con num2 y se devuelve el mayor, que es 20.

## ***Método Min***

El método Min en C# es un método estático de la clase Math que permite obtener el número más pequeño entre dos números de cualquier tipo numérico compatible, tales como int, double, float, long, etc.

**Sintaxis:** Math.Min(valor1, valor2)

Donde valor1 y valor2 son los dos números que se desean comparar. Este método devuelve el número más pequeño entre los dos.

**Ejemplo:**

```
int num1 = 10;
int num2 = 20;
int resultado = Math.Min(num1, num2);
Console.WriteLine("El número más pequeño es: " + resultado);
```

Este código escribirá en la consola «El número más pequeño es: 10».

## ***Método Pow***

El método Math.Pow en C# permite calcular la potencia de un número. Toma dos argumentos: el primer argumento es la base de la potencia y el segundo argumento es el exponente. Devuelve el resultado de elevar la base a la potencia del exponente.

**Ejemplo:**

```
double result = Math.Pow(2, 3);
Console.WriteLine(result); // output: 8
```

En este ejemplo, se está calculando 2 elevado a la potencia de 3, dando como resultado 8.

## Método sqrt

Este método se utiliza para calcular la raíz cuadrada de un número. Toma como entrada un número decimal o doble precisión y devuelve un valor decimal o doble precisión que representa la raíz cuadrada de ese número.

### Ejemplo:

```
double num = 9;
double result = Math.Sqrt(num);
Console.WriteLine("La raíz cuadrada de " + num + " es " + result);
```

Este código imprimiría el resultado siguiente: La raíz cuadrada de 9 es 3

Es importante tener en cuenta que el método Sqrt solo puede calcular la raíz cuadrada de números positivos. Si se intenta calcular la raíz cuadrada de un número negativo, se obtendrá una excepción.

## Método abs

Este método retorna el valor absoluto de un número. El valor absoluto es el valor positivo de un número sin importar su signo. Por ejemplo, el valor absoluto de -5 es 5 y el valor absoluto de 5 es también 5.

**Sintaxis:** Math.Abs(valor);

Donde valor es el número para el cual se quiere obtener el valor absoluto. Este método puede ser utilizado con valores de diferentes tipos de datos numéricos, como int, double, float, etc.

### Ejemplo:

```
int number = -10;
int result = Math.Abs(number);
Console.WriteLine("El valor absoluto de " + number + " es: " + result);
```

La función Abs devuelve el valor absoluto del número, es decir, el número sin signo. En este caso, el número -10 se convierte en 10.

## Método Round

Redondea un número decimal a un número específico de decimales o a un número entero.

Este método toma un solo argumento, que es el número decimal que se va a redondear, y devuelve un valor double que representa el resultado del redondeo.

Hay varias sobrecargas del método Round que permiten controlar cómo se realiza el redondeo, por ejemplo, permitiendo especificar el número de decimales a los que se va a redondear o permitiendo especificar la estrategia de redondeo a utilizar en casos donde un número decimal está exactamente en el medio entre dos valores enteros.

### **Ejemplo:**

```
double num = 123.456;
int decimals = 2;
double roundedNum = Math.Round(num, decimals);
Console.WriteLine("El número " + num + " redondeado a " + decimals + " decimales es: " + roundedNum);
```

En este ejemplo, la variable num contiene un número decimal y la variable decimals contiene el número de decimales a los que se va a redondear.

La línea double roundedNum = Math.Round(num, decimals); llama al método Round y almacena el resultado en la variable roundedNum. Finalmente, se imprime el resultado en la consola.

## Clase String

La clase String proporciona miembros para comparar cadenas, probar cadenas para la igualdad, buscar caracteres o subcadenas en una cadena, modificar una cadena, extraer subcadenas de una cadena, combinar cadenas, dar formato a valores, copiar una cadena y normalizar una cadena.

Entre los métodos mas comunes de la clase encontramos:

- |                    |                      |                       |
|--------------------|----------------------|-----------------------|
| ❖ CompareTo        | ❖ Insert             | ❖ StartsWith          |
| ❖ Concat           | ❖ IsNullOrEmpty      | ❖ Substring           |
| ❖ Contains         | ❖ IsNullOrWhiteSpace | ❖ ToLower             |
| ❖ EndsWith         | ❖ LastIndexOf        | ❖ ToUpper             |
| ❖ EndsWith         | ❖ Length             | ❖ Trim                |
| ❖ Equals           | ❖ Remove             | ❖ TrimStart y TrimEnd |
| ❖ EqualsIgnoreCase | ❖ Replace            | ❖ ToString            |
| ❖ IndexOf          | ❖ Split              |                       |

## ***CompareTo***

Sirve para comparar cadenas, devuelve un número según el resultado. Recuerda que no sigue el alfabeto español, lo compara según la tabla ASCII.

```
string cadena1 = "Manzana";  
string cadena2 = "Manzana";  
int resultado = cadena1.CompareTo(cadena2); // Retorna 0, ya que son iguales.
```

## ***Concat***

Concatena dos cadenas, es como el operador +.

```
string cadena1 = "Hola";  
string cadena2 = " Mundo";  
string resultado = string.Concat(cadena1, cadena2); // Retorna "Hola Mundo"
```

## ***Contains***

Puede especificar opciones de comparación para buscar una subcadena.

```
string cadena = "Este es un ejemplo";  
bool contiene = cadena.Contains("ejemplo"); // Retorna true
```

## ***EndsWith***

Indica si la cadena acaba con el String pasado por parámetro

```
string cadena = "Hola Mundo";  
bool terminaCon = cadena.EndsWith("Mundo"); // Retorna true
```

## ***Equals***

Indica si una cadena es igual que otra.

```
string cadena1 = "Hola";  
string cadena2 = "Hola";  
bool sonIguales = cadena1.Equals(cadena2); // Retorna true
```

## ***IndexOf***

Encuentra la posición de la primera aparición de una subcadena.

```
string cadena = "Esta es una cadena de ejemplo";  
int indice = cadena.IndexOf("es"); // Retorna 5
```

## ***Insert***

Inserta una subcadena en una posición específica

```
string cadena = "Hola Mundo";  
string resultado = cadena.Insert(5, "hermoso "); // Retorna "Hola hermoso Mundo"
```

## ***IsNullOrEmpty***

Comprueba si la cadena es nula o vacía.

```
string cadena = "";  
bool esNulaOVacia = string.IsNullOrEmpty(cadena); // Retorna true
```

## ***IsNullOrWhiteSpace***

Comprueba si la cadena es nula o solo contiene espacios en blanco.

```
string cadena = " ";  
bool esNulaOBlanco = string.IsNullOrWhiteSpace(cadena); // Retorna true
```

## ***LastIndexOf***

Encuentra la posición de la última aparición de una subcadena.

```
string cadena = "Este es un ejemplo, y otro ejemplo.";
int indice = cadena.LastIndexOf("ejemplo"); // Retorna 26
```

## ***Length***

Devuelve la longitud de la cadena.

```
string cadena = "Hola, mundo";
int longitud = cadena.Length; // Retorna 11
```

## ***Remove***

Elimina un número especificado de caracteres desde una posición dada.

```
string cadena = "Esto es una prueba";
string resultado = cadena.Remove(5, 3); // Retorna "Esto una prueba"
```

## ***Replace***

Devuelve un String cambiando los caracteres que nosotros le indiquemos

```
string cadena = "Hola, Hola, Hola";
string resultado = cadena.Replace("Hola", "Adiós"); // Retorna "Adiós, Adiós, Adiós"
```

## ***Split***

Divide la cadena en un arreglo de subcadenas usando un separador

```
string cadena = "Manzana,Plátano,Uva";
string[] frutas = cadena.Split(',');
// Retorna un arreglo ["Manzana", "Plátano", "Uva"]
```

## ***StartsWith***

Indica si la cadena empieza por una cadena pasada por parámetro

```
string cadena = "Hola Mundo";  
bool comienzaCon = cadena.StartsWith("Hola"); // Retorna true
```

## ***Substring***

Trocea un String desde una posición a otra.

```
string cadena = "Esto es una cadena de ejemplo";  
string subcadena = cadena.Substring(5, 10); // Retorna "es una cade"
```

## ***ToLower***

Convierte el String a minúsculas.

```
string cadena = "Hola Mundo";  
string enMinusculas = cadena.ToLower(); // Retorna "hola mundo"
```

## ***ToUpper***

Convierte el String a mayúsculas.

```
string cadena = "Hola Mundo";  
string enMayusculas = cadena.ToUpper(); // Retorna "HOLA MUNDO"
```

## ***Trim***

Elimina los espacios del String.

```
string cadena = "  Espacios en blanco  ";  
string sinEspacios = cadena.Trim(); // Retorna "Espacios en blanco"
```

## ***TrimStart y TrimEnd***

Elimina espacios en blanco o caracteres específicos al principio o al final.

```
string cadena = "----Texto----";  
string sinGuionesAlInicio = cadena.TrimStart('-'); // Retorna "Texto----"  
string sinGuionesAlFinal = cadena.TrimEnd('-'); // Retorna "----Texto"
```

## ***ToString***

Convierte la cadena en su representación de cadena.

```
int numero = 42;  
string cadena = numero.ToString(); // Convierte el número en la cadena "42"
```