

Instalación del entorno de desarrollo

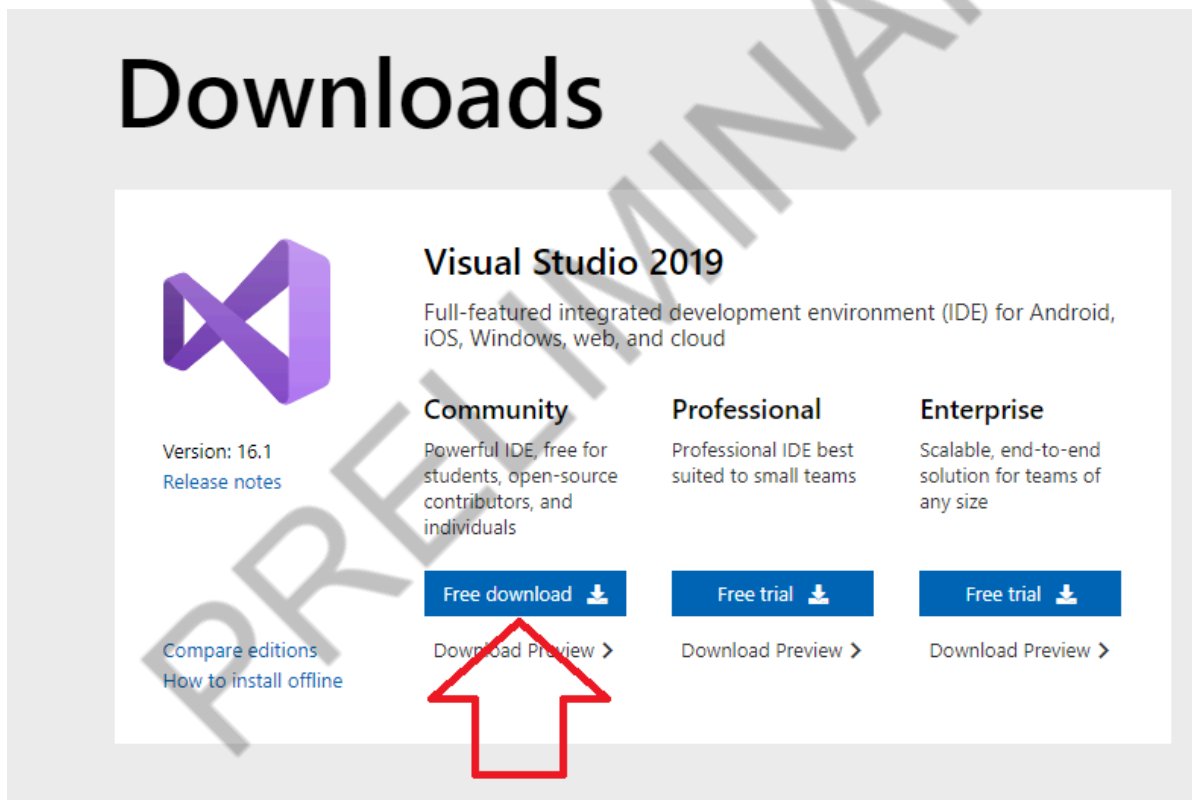
1 - Descargar e instalar Visual Studio

En esta primera entrada empezaremos por lo más básico pero a la vez más importante. consiste en instalar y poner en funcionamiento nuestro entorno de desarrollo.

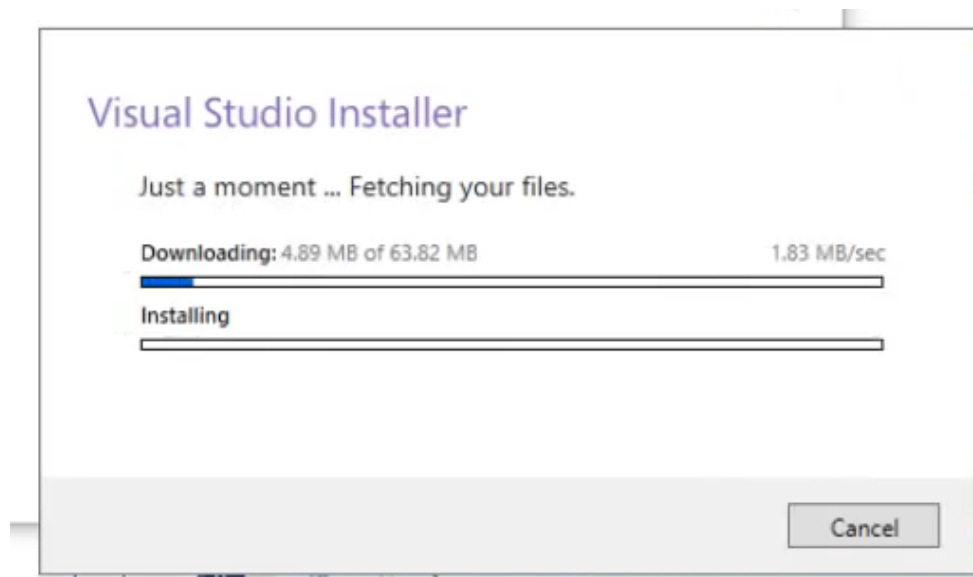
Un entorno de desarrollo es aquella aplicación que utilizamos para programar.

En concreto para programar en .Net utilizaremos visual studio, el cual lo podemos [descargar desde aquí en español de manera gratuita y oficial](#)..

Una vez en el enlace, descargamos la versión comunidad.



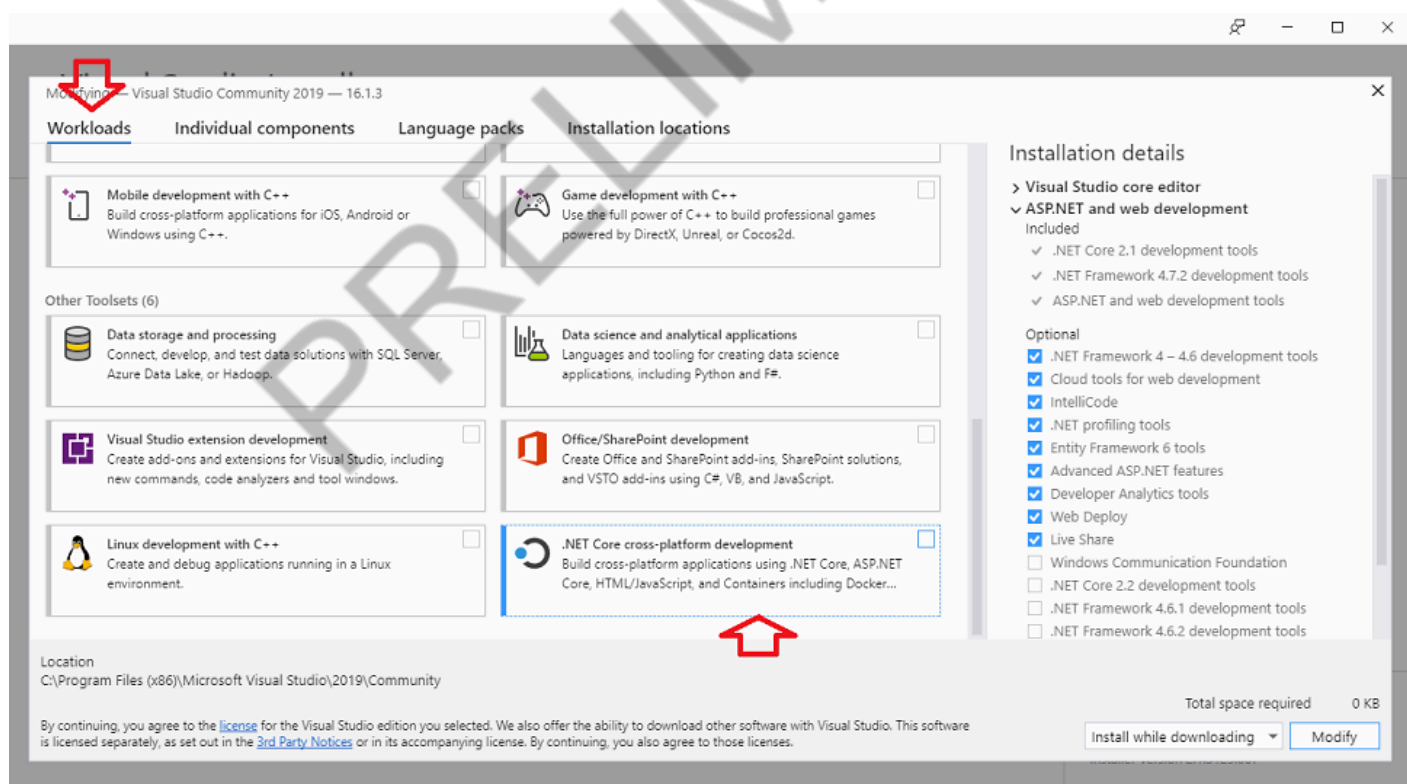
Lo que nos descarga es el launcher del instalador. por lo tanto, una vez descargado lo tenemos que abrir. y nos descargara el instalador



Una vez finalice la instalación, se nos abrirá automáticamente el instalador de visual studio. Si por ejemplo tuviésemos la versión de 2017 junto con la de 2019 se nos juntarían ambas en un solo instalador.

La ventana que veremos es como la siguiente, la cual contiene una serie de componentes que podemos añadir a nuestro visual studio.

estos componentes los podemos añadir tanto individualmente, como por paquetes. lo cual es mucho más cómodo.



Para nosotros solo es necesario uno de los paquetes, el que incluye ".Net Core." y pulsamos en instalar.

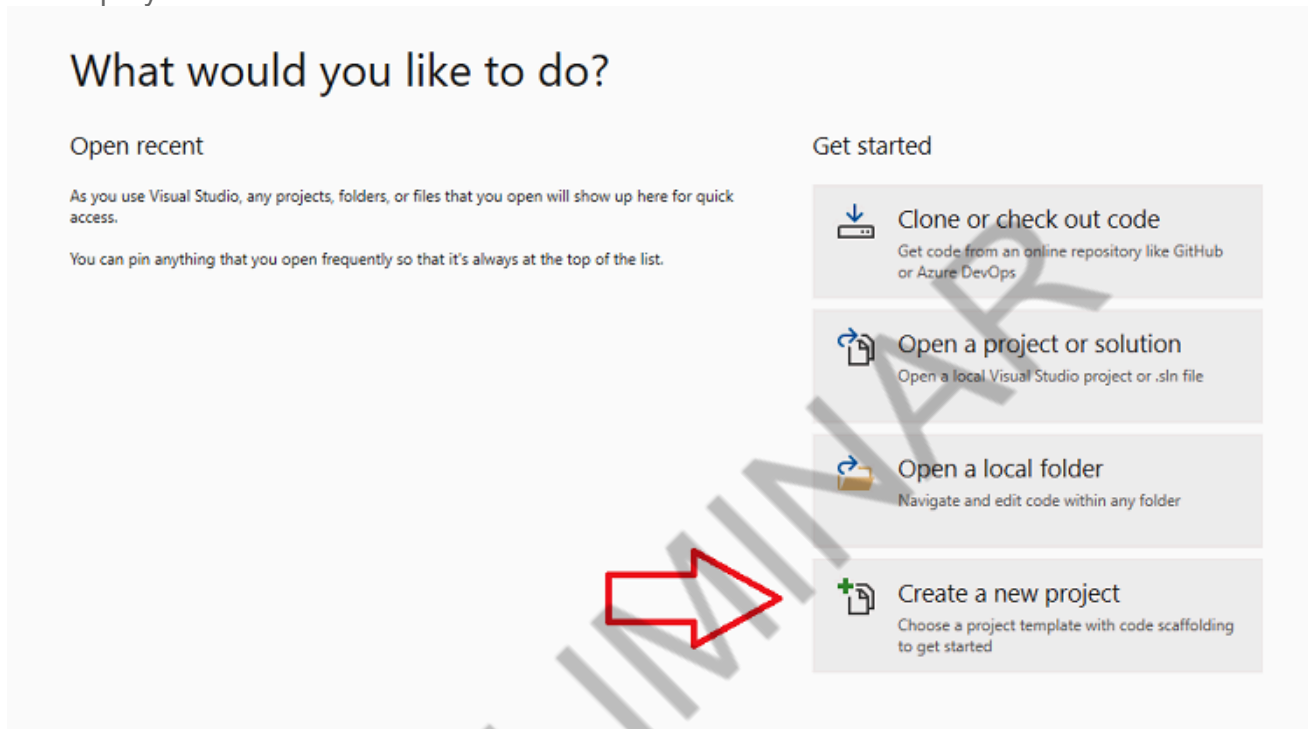
El tamaño de la descarga es más o menos grande, unos 7Gb por lo que puede tardar un rato. cuando finalice, veremos que en el mismo launcher tendremos un botón que pone

"Lanzar" al hacer click sobre el (o buscarlo en el menú de inicio) se nos abrirá visual studio 2019.

2 - Creación de una aplicación en Visual Studio

En la siguiente ventana tendremos varias opciones

- Clonar código (de un repositorio)
- Abrir un proyecto o solución
- Abrir una carpeta
- Crear un proyecto nuevo



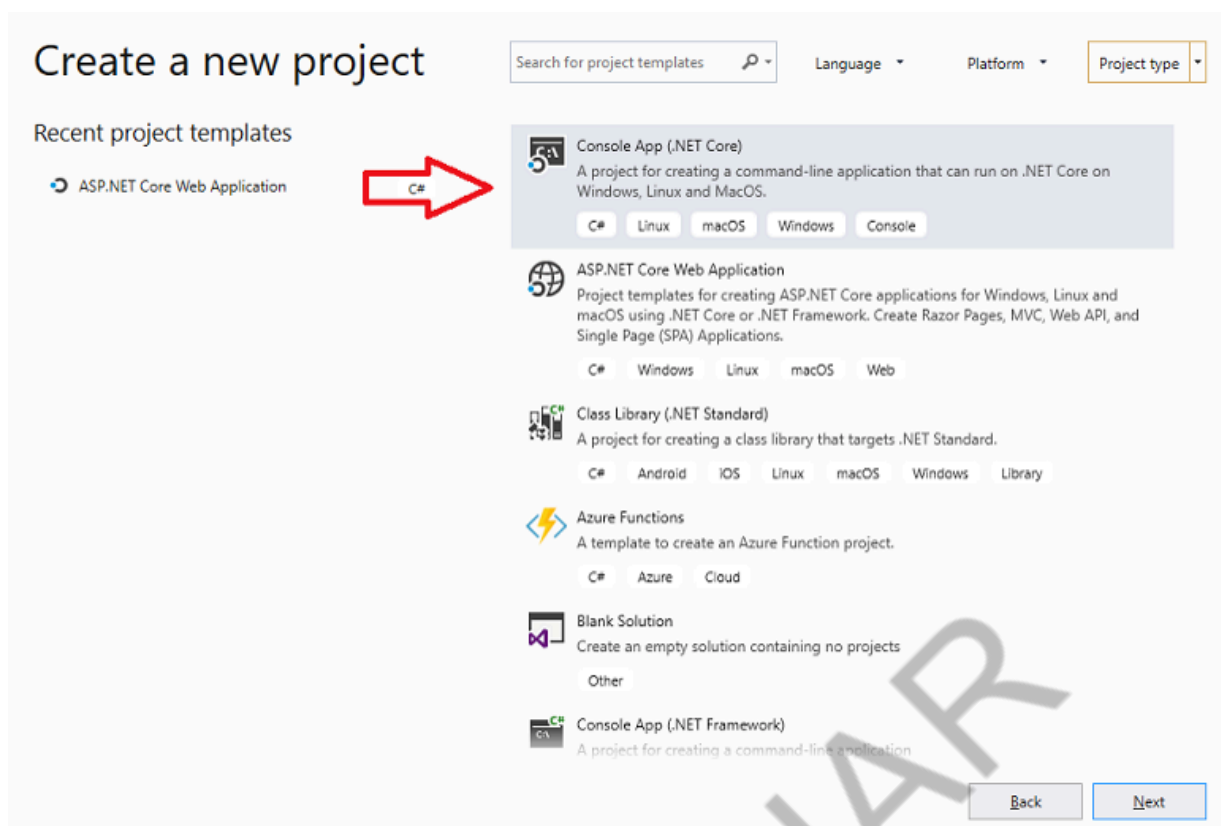
En nuestro caso la opción que nos interesa es la de crear un proyecto nuevo.

Al pulsar en ella se nos abrirá otro menú en el cual nos dejará introducir que tipo de proyecto queremos.

en esta venta podemos encontrar varios filtros

- Tipo de proyecto reciente utilizado
- Lenguaje del proyecto (en nuestro caso utilizaremos c#)
- Plataforma compatible con el tipo de proyecto
 - Ya que no es lo mismo programar para android que para IOs y todo ello lo podemos hacer desde visual studio
- Tipo de proyecto
 - Permite crear tanto aplicaciones móviles, como de escritorio, IoT, y muchas mas

Nosotros tendremos que indicar la primera opción. Aplicación de consola (.NET Core)



Pulsamos en siguiente y nos indicará que indiquemos un nombre para el proyecto. La carpeta donde se va a encontrar, yo personalmente recomiendo poner una carpeta en C:\ que se llame proyectos, y ahí todos los proyectos en los que se vaya a trabajar. y finalmente un nombre para la solución. Comúnmente el mismo que para el proyecto.

Y ya esta, esto nos creara nuestra primera aplicación de consola.

Por qué .NET Core

Hay varios motivos por los que recomiendo utilizar .NET Core. El principal y más importante es porque corre en todas las plataformas, tanto Windows, como iOS, como Linux, esta web mismo está programada en .NET Core y corriendo en un servidor Linux - prepararé un vídeo para mostrar los pasos de como hacerlo -

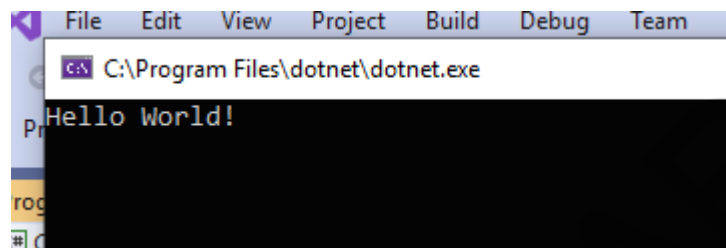
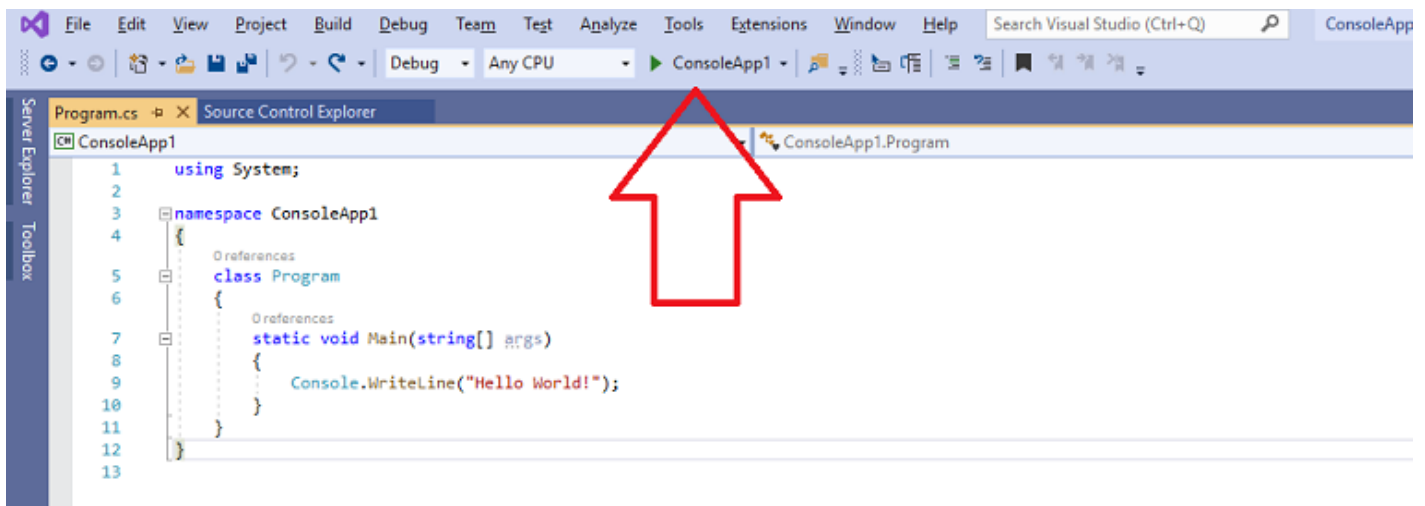
Otro punto muy importante, almenos para mi es el IDE o entorno de desarrollo, que es Visual Studio el cual no tiene rival alguno en ningún otro lenguaje de programación.

3 - Primera aplicación "Hello world"

Como podemos observar por defecto Visual Studio nos da un "hello world"

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
}
```

Si pulsamos en el botón de "Play" en nuestro visual studio nos arrancará la aplicación y nos imprimirá un "hello world"



Así de simple y sencillo es ejecutar nuestro primer "Hello world".

Antes de avanzar hay que indicar que cuando creamos un nuevo proyecto de consola siempre va a crearse una clase llamada Program y un método Main como los principales, estos se pueden cambiar, pero está estandarizado el no hacerlo.

4 - Estructura de un programa

Para crear una clase tenemos que :

pulsar botón secundario en el proyecto > añadir nuevo ítem > seleccionar clase (class)

Y nos creara algo muy similar a lo siguiente:

```
class Vehiculo{
}
```

Las clases contienen como elementos principales métodos y propiedades pero además pueden contener events, delegates, u otras clases dentro de otras, pero esto lo veremos más adelante.

4.1 - Propiedades

Las propiedades son atributos que contienen un valor a nivel de clase.

```
class Vehiculo{
    public decimal VelocidadMaxima { get; set; }
}
```

como vemos el ejemplo la creación de la propiedad continente:

- Modificador de acceso "Public"

- Tipo de dato “Decimal”
- Nombre de la propiedad “VelocidadMaxima”
- para dar valor un “set”
- para leer el valor un “get”
-

4.2 - Métodos

Los métodos son bloques de código los cuales tienen una finalidad que es realizar acciones.

El método más común en todas las clases es el constructor el cual se escribe de la siguiente manera:

```
class Vehiculo{
    public decimal VelocidadMaxima { get; set; }

    public Vehiculo(decimal velocidadMaxima){
        VelocidadMaxima = velocidadMaxima;
    }
}
```

Como vemos se llama igual que la clase. Además de esto los constructores se utilizan para dar un valor a las propiedades que contiene la clase.

Para dar un valor a las propiedades tenemos varias opciones:

- Asignar el valor dentro del constructor directamente
- Pasar el valor al constructor
- Utilizar otro método para asignar el valor
- Asignar el valor de forma directa una vez la clase esta instanciada

En nuestro ejemplo utilizaremos el de pasar al constructor el valor.

Para el ejemplo crearé otra variable que diga el consumo por kilómetro.

Esto nos permitirá crear otro método que nos devolverá el consumo total en cierta cantidad de kilómetros.

```
class Vehiculo{
    public decimal VelocidadMaxima { get; set; }
    public decimal ConsumoPorKilometro { get; set; }

    public Vehiculo(decimal velocidadMaxima, decimal consumoPorKilometro){
        VelocidadMaxima = velocidadMaxima;
        ConsumoPorKilometro = consumoPorKilometro;
    }

    public decimal ConsumoTotal(decimal kilometros){
        return ConsumoPorKilometro * kilometros;
    }
}
```

Como vemos en este caso es algo diferente ya que el método tiene después de la palabra “public” otro tipo de dato, el cual es el valor de retorno del método.

Importancia de los nombres de las variables

Es importante hacer un pequeño inciso aquí, ya que es un elemento que normalmente los programadores junior no caen en cuenta y es el nombre de las propiedades/variables y los métodos.

Es importante que tanto propiedades como métodos tengan nombres que se auto explican, y con esto que quiero decir, que si llamamos a una propiedad "VelocidadMaxima" todo el mundo cuando lee el código un tiempo después sabe que ahí en esa propiedad está almacenada la velocidad máxima. En cambio si llamamos a la variable "xmx" nadie va a entender que es lo que hace esa propiedad

4.3 - Comentarios

Los comentarios son bloques de texto añadidos a mano, y estos son añadidos para explicar qué es lo que pasa en el código, si alguna parte del mismo es muy compleja o liosa nos veremos en la obligación de poner algunos. pero como he indicado hace un momento, lo ideal sería que cada nombre de cada variable, propiedad y método nos indique claramente de lo que se trata.

Hay 3 formas de añadir comentarios:

- Comentarios de una línea utilizando "//"
- Bloque de comentario "/* texto */"
- Descripción de una función "///"
 - al pulsar 3 veces contrabarra si estamos en la parte superior de una función, el IDE nos generará automáticamente un bloque de comentarios el cual contiene:
 - <summary> para explicar qué es lo que sucede en la función
 - <param> para cada uno de los parámetros de entrada
 - <returns> para indicar que es lo que devuelve

```
//comentario de línea

/*
    Comentario de bloque
*/

/// <summary>
/// Indica la cantidad de litros de gasolina gastados
/// </summary>
/// <param name="kilometros">Indicar kilometros realizados</param>
/// <returns></returns>
```

Como podemos observar, si tanto variables como métodos tienen un nombre adecuado, la utilización de comentarios no es necesaria dado que nos dan información que ya tenemos.

4.4 - Modificadores de acceso

Este apartado lo vamos a ver muy por encima ya que realizare otro video en el cual lo explicare todo con más detalle, simplemente enumerarlos y nombrar sus características.

Un modificador de acceso como su propio nombre indica, permite indicar la accesibilidad que tienen nuestros métodos o propiedades desde otros miembros o clases que los referencian.

Los principales son:

- public, sin restricciones
- internal, solo se puede acceder a ellos desde el mismo proyecto
- private, solo accesible desde la clase en la que se genera
- protected, desde la propia clase y desde las derivada.

por defecto en .NET el modificador de acceso es “internal”

4.5 -Declaración del namespace

Qué es el namespace o espacio de nombres?

el namespace es el lugar donde organizamos todas las clases y ficheros de nuestro proyecto. por detrás este usa un lenguaje XML el cual contiene qué clases están dentro de cada namespace y así permite usarla sin utilizar la directiva “using” la cual permite utilizar clases dentro de otros namespaces. (esto es muy utilizado en el día a día).

La utilización de namespace nos permite esquivar la ambigüedad con los nombres de las clases, ya que no puede haber dos clases con el mismo nombre en el mismo namespace, pero si en el mismo proyecto en diferentes namespaces.

Sentencias de toma de decisiones

Ha llegado la hora de introducir uno de los puntos más importantes de toda la programación. La toma de decisiones, cuando realizar una acción u otra, y como realizar esta acción, si tiene que ser en bucle o una condición nos indicara si realizarla.

Índice

- 1 - Sentencias Condicionales
 - 1.1 - If -else
 - 1.2 - Swich-case
- 2 - Sentencias Iteracionales - Bucles
 - 2.1 - Bucle For
 - 2.2 - Bucle while
 - 2.3 - Bucle do-while

1 - Sentencias Condicionales

Son las que realizamos cuando tenemos que tomar una decisión basándonos en cierta lógica, una forma de verlo gráficamente sería. Imaginémonos que vamos por una carretera y esa carretera tiene un desvío, si tomamos el desvío sería como entrar en una sentencia condicional.

1.1 - If -else

La primera sentencia condicional que vamos a ver es la sentencia **If-Else** la cual literalmente significa "sí-sino" por lo tanto la sentencia condicional va enlazada con un **operador lógico** y se representa de la siguiente forma

```
if (A < B)
{
    Console.WriteLine("A es menor que B");
}
else
{
    Console.WriteLine("A no es menor que B");
}
```

Si el contenido dentro de la sentencia if es una sola línea, no necesitamos utilizar los paréntesis, pero personalmente lo recomiendo. Ya que así es más fácil leerlo después.

```
if (A < B)
    Console.WriteLine("A es menor que B");
```

```
else
    Console.WriteLine("A no es menor que B");
```

Desde el momento de escribir este artículo, he estado pensando en hacer que te pierdas si estas leyendo el código. En el trabajo he tenido varios casos en los que tardas mas tiempo en saber que hace tanto if dentro de tanto if que en arreglar el propio problema en sí.

```
A < B ? Console.WriteLine("A es menor que B") : Console.WriteLine("A no es menor que B");
```

En caso de que solo quieras incluir la opción de dentro del `if` porque el `else` va a seguir el curso normal de la ejecución no necesitas añadir el `else`.

```
if (A < B)
{
    Console.WriteLine("A es menor que B");
}
```

1.2 - Switch-case

Muchas veces nos encontramos con que escribimos muchas sentencias IF seguidas una detras de otra. para solucionar este problema tenemos la implementación de las sentencias switch.

una sentencia switch coge una variable y la analiza por cada uno de los casos disponibles, como vemos en el ejemplo indicamos `switch(variable)` para analizar la variable. y esta puede ser de cualquier tipo, no tiene porque ser un string, puede ser un numero, una fecha, etc.

```
string mes = "enero";
switch (mes)
{
    case "enero":
        Console.WriteLine("enero tiene 31 dias");
        break;
    case "febrero":
        Console.WriteLine("febrero tiene 28 dias");
        break;
    default:
        Console.WriteLine("mes no encontrado");
        break;
}
```

Como vemos en el ejemplo, analizamos cada case y al final del mismo introducimos la sentencia break; la cual nos hara salir del bucle. Para este ejemplo hemos utilizado únicamente dos meses, con lo que la sentencia es bastante pequeña pero en caso de poner todos los meses, sería mucho mayor.

Como todo el mundo sabe, los meses (a excepción de febrero) tienen 30 o 31 días, la sentencia switch nos permite agrupar en caso de que queramos la misma salida para varios de ellos

```
string mes = "enero";
switch (mes)
{
    case "enero":
    case "marzo":
    case "mayo":
    case "julio":
    case "agosto":
    case "octubre":
    case "diciembre":
        Console.WriteLine("enero tiene 31 dias");
        break;
    case "febrero":
        Console.WriteLine("febrero tiene 28 dias");
        break;
    case "abril":
    case "junio":
    case "septiembre":
    case "noviembre":
        Console.WriteLine("febrero tiene 28 dias");
    default:
        Console.WriteLine("mes no encontrado");
        break;
}
```

Con C# 8 su funcionalidad ha aumentado, puedes darles una ojeada [aquí](#).

2 - Sentencias Iteracionales - Bucles

Comúnmente cuando estamos programando nos encontramos con que tenemos una lista de objetos y a todos ellos (o a una mayoría aplicando sentencias condicionales) debemos calcular algún valor o imprimir al usuario distinta información. para ello en vez de hacer cada caso uno por uno, utilizamos bucles, que nos permiten iterar sobre cada uno de los elementos de una lista.

2.1 - Bucle For

El bucle for es el bucle mas implementado, y se utiliza para reptir acciones cierto numero de veces

```
for (int i = 0; i < 10 ; i++){  
    Console.WriteLine("Iteración número "+i);  
}
```

Como vemos cuando declaramos el bucle for, lo hacemos junto con:

- Declaración e inicialización de la variable: `int i = 0;`
- Expresión lógica para comparar si se tiene que ejecutar: `i < 10;`
- Operador aritmético de incremento en una unidad: `i++;`

2.2 - Bucle while

Evaluamos la expresión lógica entre paréntesis y ejecutamos el bloque si es verdadera, al terminar volvemos a evaluar la expresión, y si es válida volvemos a ejecutar el bloque.

```
int contador = 0;  
while(contador < 10){  
    Console.WriteLine("Iteración número "+i);  
    contador++;  
}
```

2.3 - Bucle do-while

Similar a la anterior, solo que en este caso el bloque se ejecuta al menos una vez y posteriormente sigue ejecutando mientras la expresión sea cierta.

```
int contador = 0;  
do{  
    Console.WriteLine("Iteración número "+contador);  
    contador++;  
}while(contador < 10);
```

Hay que tener mucho cuidado cuando creamos bucles ya que es posible -sobre todo cuando estamos aprendiendo - hacer bucles infinitos. los cuales, si suceden en nuestra aplicación cuando estamos desarrollando, nos dará un "out of memory" porque el ordenador se queda sin memoria y ya está, no pasa nada.

pero si lo realizamos en producción, causaría que la aplicación dejará de funcionar y será un problema mucho mayor.

una forma de crear un bucle infinito es la siguiente:

```
int contador = 11;  
do{
```

```
Console.WriteLine("Iteración número "+contador);  
    contador++;  
}while(contador > 10);
```

Corrección: el bucle no dejará de ejecutarse nunca. Obtendremos el mismo resultado si por ejemplo no pusieramos la sentencia `contador++` , ya que la variable contador tendría siempre el mismo valor.

PRELIMINAR

Variables y operadores

Índice

- 1 - Qué son las variables
 - 1.1 - Tipos de variable
 - 1.2 - Declaración de variables
 - 1.3 - Tipo implícito de variables
 - 1.4 - Declaración de constantes
- 2 - Qué son los operadores
 - 2.1 - Operadores aritméticos.
 - 2.2 - Operadores relacionales
 - 2.3 - Operadores lógicos
 - 2.4 - Operadores de asignación

1 - Qué son las variables

Lo primero que tenemos que tener en cuenta es qué son las variables. Son nombres que apuntan a un valor que el compilador toma de forma dinámica. Cada variable necesita tener un tipo el cual determina su tamaño y la forma en la que es almacenado en memoria

1.1 - Tipos de variable

Disponemos de multitud de tipos de variables, en el video solo enumero las principales, ya que los videos estan enfocados al mundo real, y en el mundo real tan solo utilizamos unas pocas aquí enumerare alguna más.

Tipo	Descripción
int, long, sbyte, byte, short, ushort, uint, ulong, char	Números Enteros
Float, doble	Números de coma flotante
Decimal	Números decimales
Bool	verdadero o falso
null	tipos nulos

Además de estos tipos C# permite tener enum y referencias tipo de variable y que es muy común utilizar variables de tipo de referencia como pueden ser las clases, interfaces, o delegados en el mundo real, ya que al trabajar se intenta hacer lo más similar al mundo real posible, lo cual nos lo facilita mucho. Los tipos de referencia los veremos más adelante.

1.2 - Declaración de variables

Declarar una variable es muy sencillo, únicamente tenemos que indicar el tipo de variable y su nombre.

```
int ejemploVariable;
```

Pero en este ejemplo la variable no tienen ningún valor, si intentáramos consultarla nos diría que es `null` por lo que es como si no tuviésemos nada y si fuéramos a operar con ella el compilador nos

daria un error. Para evitar esto, debemos asignarle un valor.

para ello tenemos dos formas, asignarle el valor una vez está creada, o asignarle el valor en la propia creación de la variable.

```
int ejemploVariable;  
ejemploVariable = 5;  
  
//Segunda forma  
  
int ejemploVariable = 5;
```

Ambos ejemplos nos crearan una variable `ejemploVariable` con un valor numérico de 5. Personalmente prefiero la segunda forma. Si utilizamos la primera, el compilador nos dirá que qué tal si utilizamos la segunda, que es mejor para que la lean los humanos.

Podemos definir mas de una variable si las separamos porp coma (,) y hay que poner punto y coma (;) al terminar cada sentencia.

```
int ejemploVariable, ejemploVariable2;
```

1.3 - Tipo implícito de variables

Desde la entrada de C# 3 podemos utilizar el tipo implícito para crear una variable.

Esto que quiere decir, que no tenemos porqué indicar que tipo de variable es cuando la declaramos ya que cogerá el tipo de variable de la inicialización.

```
var ejemploVariable = 5;
```

Como vemos en el ejemplo utilizamos la palabra reservada `var` la cual se le asigna el tipo del valor que asignamos a la variable.

Esto se puede hacer tanto con tipos primitivos como con tipos de referencia creados por nosotros mismos. Personalmente recomiendo utilizar el tipo de variable en vez de var ya que a la hora de leer el codigo, o hacer un "code review" ([curso git y github](#)) es mucho mas sencillo de entender.

1.4 - Declaración de constantes

Algunas veces solo tenemos que asignar el valor a una variable una vez ya que este no va a cambiar en toda la ejecución del programa. Para ello utilizaremos lo que se denomina como constante. Ello nos proporciona ciertas ventajas sobre las variables convencionales.

- Nos aseguramos que ese código no cambia; en un proyecto pequeño nos puede dar mas igual, pero en uno grande puede llegar a ayduar bastante.
- Debido al punto anterior el mantenimiento, o si falla es mas sencillo, ya que no tenemos que ir buscando lugares en los que el valor cambia.
- El compilador es mas eficiente; Ya que no tiene que estar preocupandose de si el valor cambia o no, por lo que puede optimizar el código a sabiendas de que el valor siempre es el msimo.

Un ejemplo de consante podría ser el siguiente, recordad que es un valor que no cambia nunca.

```
public const int numeroMeses = 12;
```

2 - Qué son los operadores

Ahora que hemos visto que son las variables pasaremos a otro punto de vital importancia, los operadores. Los cuales nos permiten realizar operaciones tanto matemáticas con operadores como lógicas.

C# tiene una gran variedad de operadores como vamos a ver a continuación.

2.1 - Operadores aritméticos.

Son aquellos operadores que nos permiten realizar acciones, normalmente matemáticas:

Operador	Descripción	Ejemplo
+	Suma o concatena dos operadores.	$A + B = 10$
-	Resta el segundo operador al primero.	$A - B = 5$
*	Multiplica ambos operadores.	$A * B = 25$
/	Divide ambos operadores.	$A / B = 1$
%	Operador módulo, nos devuelve el resto de la operación.	$A \% B = 0$
++	Incrementa el valor en una unidad.	A++ devuelve 6
--	Decrementa el valor en una unidad.	A-- devuelve 4

Esto es un pequeño ejemplo, si cambiamos el operador con cualquiera de los de arriba veremos como nos da la salida correspondiente.

```
int operadorA = 5;
int operadorB = 5;
int resultado = operadorA + operadorB;
//Resultado nos devolvera 10 en este caso.
```

2.2 - Operadores relacionales

Los operadores relacionales son aquellos que nos permiten realizar una comparación; Supongamos que los numeros a comparar son $A = 5$, $B = 2$:

Operador	Descripción	Ejemplo
==	Comprueba si ambos operadores son iguales.	$(A == B)$ devuelve falso
>	Comprueba que el valor de la izquierda es mayor que el de la derecha. Si lo es, el valor es verdadero.	$(A > B)$ devuelve verdadero

<	Comprueba que el valor de la izquierda es menor que el de la derecha. Si lo es, el valor es verdadero.	(A < B) devuelve falso
>=	Comprueba que el valor de la izquierda es mayor o igual que el de la derecha. Si lo es, el valor es verdadero.	(A > B) devuelve verdadero
<=	Comprueba que el valor de la izquierda es menor o igual que el de la derecha. Si lo es, el valor es verdadero	(A < B) devuelve falso
!=	Comprueba que ambos valores son iguales o no. Si no son iguales devuelve verdadero	(A != B) devuelve verdadero

2.3 - Operadores lógicos

Son las expresiones lógicas y comparan booleanos; Para el ejemplo A = true, B = false :

Operador	Descripción	Ejemplo
&&	Operador lógico AND, si ambos booleanos son verdadero el resultado será verdadero.	(A && B) devuelve falso
	Operador lógico OR, si alguno de los dos es verdadero el resultado será verdadero.	(A B) devuelve verdadero
!	Operador lógico NOT, usado para negar una operación lógica, si una condición es verdadera entonces el operador NOT la convertirá en falsa.	!(A && B) devuelve verdadero

2.4 - Operadores de asignación

Son los operadores que utilizamos para asignar valor a una variable.

Operador	Descripción	Ejemplo
=	Operador de asignación. Asigna un valor desde el lado derecho a la parte izquierda de la operación.	C = A + B ; Asigna el valor de A + B dentro de C
+=	Operador Aritmético de añadir y operador de asignación; Añade el valor de la izquierda al de la derecha y lo suma. Esta operación se puede hacer con cualquier operador aritmético.	B += A es equivalente a B = B + A

Entrada salida por teclado y pantalla

Índice

- 1 - Entrada por teclado
- 2 - Salida por pantalla

1 - Entrada por teclado

Para introducir información por teclado lo que tenemos que hacer es escribir la siguiente línea de código

```
Console.ReadLine();
```

Cuando hacemos esto, si pulsamos el botón de play en visual studio nos dejará introducir cualquier valor hasta que introducimos enter.

Pero claro, si solo recibimos la información no hacemos nada, tenemos que almacenarla en una variable

```
String entradaTeclado = Console.ReadLine();
```

2 - Salida por pantalla

Mostrar información es muy sencillo, tan solo tenemos que poner la sentencia `Console.WriteLine(variable)`

Por lo tanto, podemos hacer un ejemplo que pida al usuario su nombre y lo muestre por pantalla.

```
Console.WriteLine("Introduce tu nombre:");  
string nombre = Console.ReadLine();  
Console.WriteLine($"El nombre es {nombre}");  
Console.ReadKey();
```

La pieza de código que vemos para mostrar en el `Console.WriteLine()` se llama **string interpolation** que veremos más adelante en este curso.

Finalmente ponemos un `Console.ReadKey()` para que la ejecución del código se detenga hasta que introducimos un solo carácter por teclado

Modo Debug

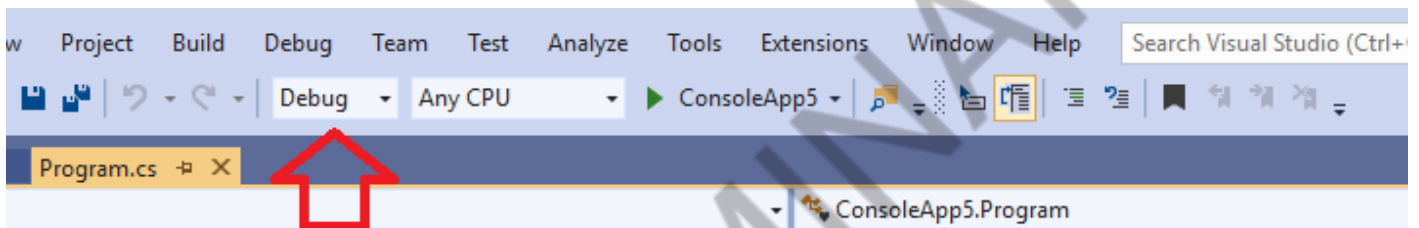
Índice

- 1 - Qué es debuguear?
- 2 - Cómo debuguear el código
- 3 - Ventajas de Debuguear

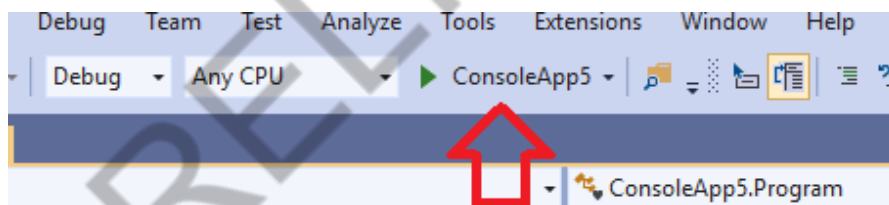
1 - Qué es debuguear?

Debuguear es lo que los desarrolladores hacemos para iterar sobre el código mientras este se esta ejecutando. Esto nos permite poder acceder a los bugs y analizarlos, ya que sin ello seria mucho mas complicado encontrar y arreglar errores.

Además, indicar que Visual Studio (o nuestro IDE) tiene que estar en el Modo Debug, en VS la opción está arriba en el menú



Para entrar a iterar el codigo lo que tenemos que hacer es pulsar el "play" que tambien vemos en la imagen previa



eApp5

Una vez estamos dentro vamos a iterar sobre el código.

```
int contador = 0;
do
{
    Console.WriteLine("Iteración número " + contador);
    contador++;
} while (contador < 10);
```

2 - Cómo debuguear el código

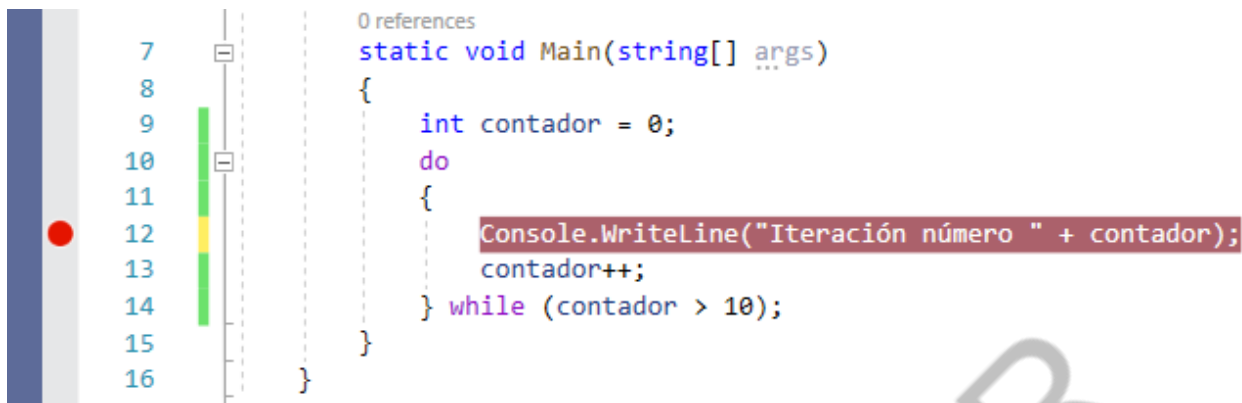
Pero para poder debuguear el código como tal necesitamos añadir `breakpoints` , o puntos de ruptura en castellano, y para añadirlos tenemos 3 opciones.

Colocar el cursor sobre la línea y pulsar **F9**

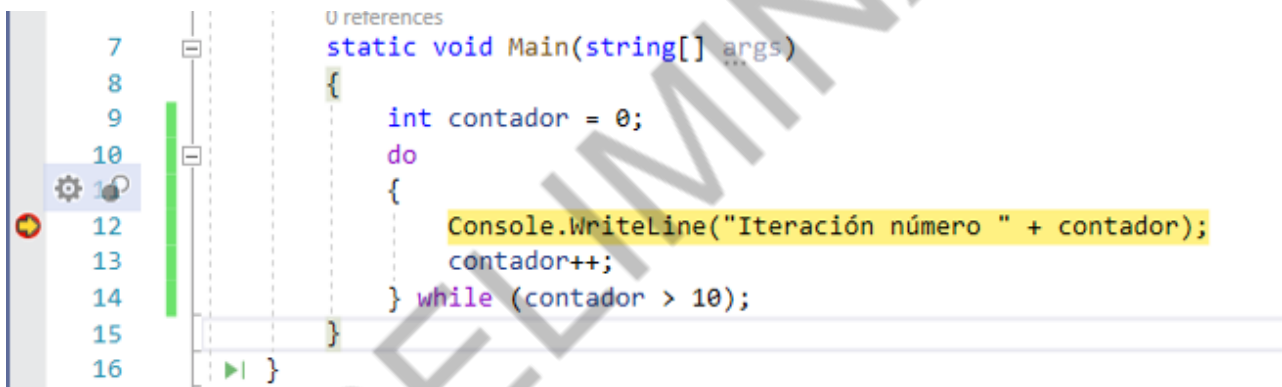
Boton secundario sobre la línea > Breakpoint > insertar breakpoint

En el lateral del fichero pulsar con el botón izquierdo del ratón, y nos añadira un breakpoint

Cuando hemos añadido un breakpoint nos saldrá un punto rojo, el cual indica que es un breakpoint (es el mismo punto que pinchamos utilizamos la opcion 3) además la linea se nos pondrá de color rojo.



Una vez el programa está en ejecución, parará automáticamente al llegar al breakpoint, y este se nos pondrá amarillo, junto con una flecha en el lateral del fichero, para indicar que es ahí el punto en el que está en este momento.



Una vez estamos aquí tenemos varias opciones para continuar:

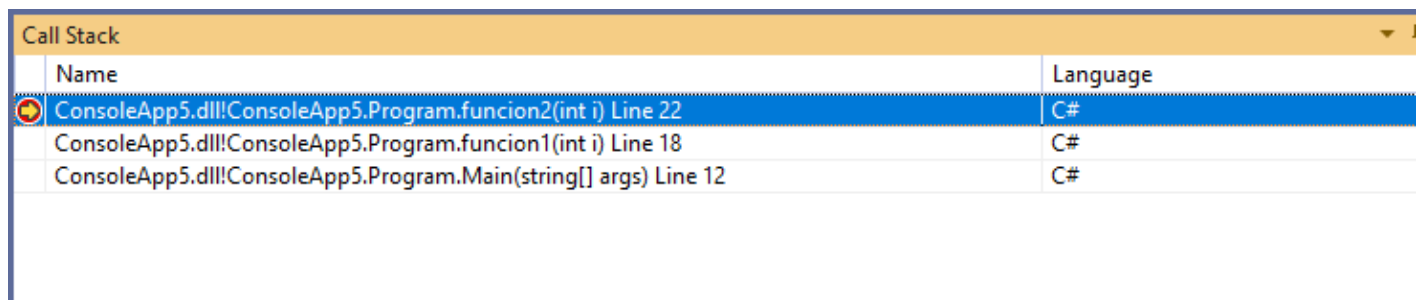
- **F5** continua hasta el siguiente breakpoint
- **F10** Ejecuta esa línea de código
- **F11** Ejecuta la siguiente sentencia de código, esto quiere decir que si por ejemplo estamos llamando a un método dentro de esa línea de código, pulsando F11 entraria dentro del método.

Otra opción que tenemos es arrastrar la flecha hasta la línea que queramos, o volver atrás, pero hay que tener cuidado ya que puede causar algunas excepciones, si algun elemento no ha sido declarado, por ejemplo.

3 - Ventajas de Debuguear

Como hemos dicho antes, podemos ver paso por paso la ejecución de un programa, y eso incluye que cuando se rompe o tenemos información corrupta podemos ver donde y porque están esos datos corruptos.

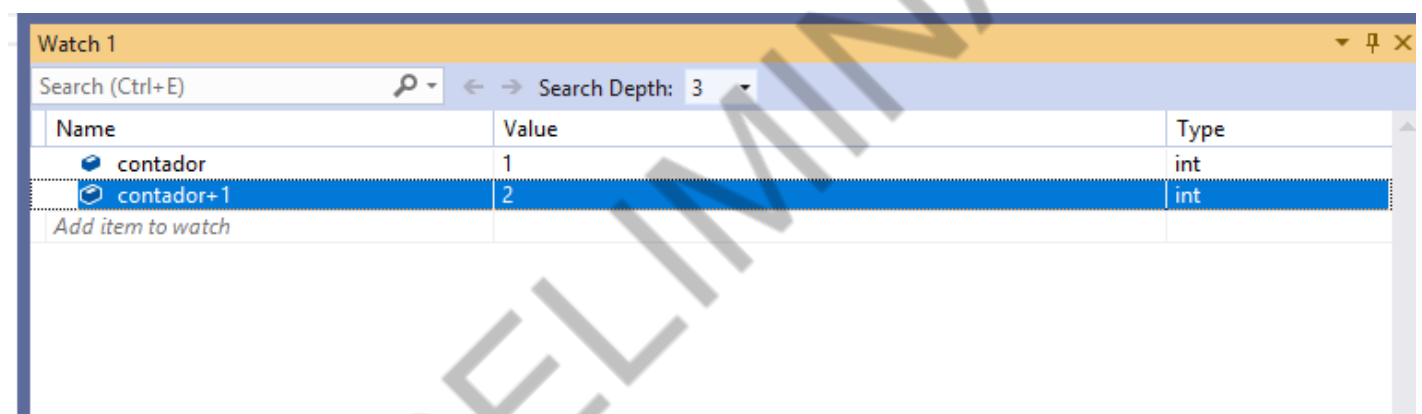
Disponemos de acceso al Stack de la llamada, que incluye cada punto por el que el proceso ha pasado:



Name	Language
ConsoleApp5.dll!ConsoleApp5.Program.funcion2(int i) Line 22	C#
ConsoleApp5.dll!ConsoleApp5.Program.funcion1(int i) Line 18	C#
ConsoleApp5.dll!ConsoleApp5.Program.Main(string[] args) Line 12	C#

Si hacemos doble click en cualquiera de ellos, Visual studio nos llevar a ese punto, pero no solo eso, las variables tendrán el valor que tenían en ese punto. Lo cual facilita mucho a la hora de comprobar errores.

Como hemos dicho cuando estamos debugueando tenemos acceso a todas las variables disponibles en el scope, por lo que podemos consultar su valor o incluso operar con ellas en la caja "watch" de visual stuido, lo cual facilita mucho cuando tenemos grandes bloques de informaición como listas de empleados etc.



Name	Value	Type
contador	1	int
contador+1	2	int
Add item to watch		

Como vemos la segunda línea es nuestro contador, que hemos operado con el, esta informacion es temporal y no es la que se va a almacenar en el código cuando volvamos a ejecutarlo.

Finalmente indicar que podemos cambiar valores de las variables cuando estamos parados en un breakpoint, eso nos puede facilitar en ciertos casos a comprobar un "y qué pasaría si..." pero personalmente no lo recomiendo.

Live Coding

- 1 – Entrar por teclado 2 números y mostrar por pantalla todos los pares
- 2 – Escribir un programa que imprima los números perfectos entre 1 y n un numero es perfecto cuando un numero es igual a la suma de sus divisores; Ejemplo $1+2+3 = 6$
- 3 – Preguntar por teclado por un nombre y repetir hasta que introduzcamos “NetMentor”
- 4 – Los 3 ejercicios anteriores tienen que estar Dentro de un menú; y entonces realizar las acciones necesarias Notas: Recuerda hacer una opción para salir del programa Utilizar funciones.

PRELIMINAR

Caracteres y cadenas de texto

Uno de los elementos más utilizados, sino el que más que mostramos de cara al usuario cuando programamos son las cadenas de texto, comúnmente llamadas frases, por lo tanto en este video vamos a ver como operar con ellos.

Índice

- 1 - El tipo char
- 2 - El tipo string
 - 2.1 - Formatear un string
 - 2.2 - Funciones de la clase string
- 3 - StringBuilder

1 - El tipo char

Lo primero que veremos es el tipo carácter, llamado tipo `char`, que es un único carácter

El cual tiene varias características:

- Son los datos predefinidos del tipo char y ocupan 1 byte en memoria.
- para indicar un carácter debemos ponerlo entre comilla simple. si utilizamos dos será considerado como una cadena.

```
char caracter = 'k';
```

esa es la forma de definir un carácter, pero en circunstancias normales no se utilizan nunca, lo que si utilizamos es el tipo string.

2 - El tipo string

Lo que si utilizamos, y mucho, son las cadenas de texto, osea varios elementos tipo char uno detrás de otro, haciendo frases. Esto se llama tipo `string`.

Hay varias formas de crear un string, la primera es directa y la más sencilla:

- Asignación de forma directa.

```
string test = "NetMentor";
```

- La segunda es utilizando un array de chars, veremos lo que son los arrays en el siguiente post.

```
char[] name = { 'N', 'E', 'T'};  
string stringName = new string(name);
```

- La tercera es concatenando dos variables

```
var nombreConcatenado = name1 + name2
```

- finalmente podemos utilizar *string interpolation*, que podemos ver más detalladamente, en otro video, pero que de primeras diremos que puede tener variables dentro siempre y cuando las introducimos entre

llaves.

```
var stringConVariables = $"el nombre es {variable}.";
```

2.1 - Formatear un string

Normalmente cuando queremos mostrar datos, suelen ser datos de un tipo, ya sea porcentaje, dinero, un formato específico de fecha o muchas otras opciones, ya que es mucho mejor mostrar 50€ que un simple 50, o el símbolo de porcentaje cuando hablamos de porcentajes

Para ello utilizamos un metodo que esta dentro de la clase string, llamado *"format"* y lo Utilizamos de la siguiente manera:

```
string.Format("La temperatura actual es {0}°C.", 20.4);
```

Como primer parámetro pasamos la frase que queremos imprimir, dentro de la frase podemos incluir variables, como hemos indicado en el punto anterior. para ello introducimos entre corchetes el número de la variable {n}, ya que después como segundo parámetro, (y siguientes) incluimos las variables que queremos enseñar.

2.2 - Funciones de la clase string

Finalmente dentro de string veremos que por defecto string trae muchas funciones de uso común para así facilitar la vida de los desarrolladores y son las siguientes:

imaginémonos que tenemos un string como el siguiente:

```
string myString = "Tengo un vaso lleno de";
```

- Si lo queremos convertir a mayúsculas:

```
string mayusculas = myString.ToUpper();
```

- Si lo queremos convertir a minuscula

```
string minusculas = myString.ToLower();
```

- Si queremos comparar una frase con la otra, nos devolverá true or false

```
bool sonIguales = myString.Equals("Tengo un vaso lleno de");
```

- Si queremos saber la posición de un carácter

```
int posicion = myString.IndexOf("o");
```

- Si queremos concatenar dos string

```
string frase = myString + " zumo de naranja";
```

- Si queremos saber si una frase contiene cierta frase, nos devuelve true falso

```
bool contiene = myString.Contains("vaso");
```


3 - StringBuilder

Cuando creamos una variable string son inmutables, lo que significa que una vez se le asigna el valor esté no se puede cambiar. Si lo que queremos es actualizar un string, internamente lo que hace el código es crear un nuevo espacio de memoria, lo cual puede causar problemas de rendimiento.

Para evitar este problema o si necesitamos una cadena de texto mutable tenemos el tipo `StringBuilder`.

```
StringBuilder sb = new StringBuilder();
```

En este caso si queremos añadir texto a la variable utilizamos el siguiente método:

```
sb.Append("texto");
```

el cual sí utiliza los mismos espacios de memoria y no crea duplicidades.

Para imprimirlo simplemente tenemos que llamar al método `.ToString()` dentro de `StringBuilder`

```
Console.WriteLine(sb.ToString());
```

Comúnmente en la vida real utilizaremos la concatenación para añadir texto, ya que los pocos bytes que gasta no compensa la lógica de crear el objeto. pero si lo que vamos a hacer es montar un texto largo, como juntar varios xml, si nos puede interesar hacer un stringBuilder para que así utilice menos espacio en memoria.

Trabajando con arrays y listas

Cuando programamos, una de las funcionalidades que más se utilizan son las colecciones de datos, ya sean estas desde una base de datos, desde un servicio externo o creadas por nosotros mismos. Por ejemplo imaginémonos que tenemos una lista de empleados o de alumnos en un colegio.

En este tutorial veremos como trabajar con este tipo de datos, no creamos una variable para cada uno de los empleados o alumnos, sino que creamos una estructura de datos llamada colecciones. Y estas colecciones pueden ser `arrays` o `Listas`.

Índice

- 1 - Qué es un Array?
 - 1.1 - Declaración de un Array
 - 1.2 - Dar valor a los elementos de un array
 - 1.3 - Trabajar con el array
- 2 - Array multidimensional
 - 2.1 - Declaración del array multidimensional
 - 2.2 - Dar valor a los elementos de un array multidimensional
 - 2.3 - Trabajar con un array multidimensional
- 3 - Jagged array
 - 3.1 - Declaración de un jagged array
 - 3.2 - Dar valor a los elementos de un jagged array
 - 3.3 - Trabajar con un jagged array
- 4 - Listas o List
 - 4.1 - Creación de una lista
 - 4.2 - Añadir elementos a una lista
 - 4.3 - Trabajar con una lista
- 5 - LINQ
 - 5.1 - Qué es LINQ
 - 5.2 - Cómo utilizar LINQ

1 - Qué es un Array?

Un array es un tipo que nos permite almacenar una colección de datos de un tipo deseado. Estos tipos pueden ser tanto primitivos como **tipos** creados por nosotros mismos.

Otra de las características que contienen los arrays es que no tienen límite de tamaño y puede contener tantos registros como queramos siempre y cuando tengamos memoria suficiente en el ordenador. Pero lo importante es que el compilador no nos pondrá un límite.

Finalmente, los arrays son Inmutables, lo que quiere decir que una vez los inicializamos estos no cambian. Por ejemplo no podemos añadir o quitar elementos de un array.

1.1 - Declaración de un Array

Para declarar un array tenemos que indicar primero de todo el tipo de dato, como hemos indicado, tanto primitivo como creado por nosotros mismos. Seguido de un corchete para abrir

y otro para cerrar `[]` que indican que es un array y cuando lo declaramos indicamos el tamaño que ese array va a tener.

```
int[] arrayEnteros = new int[5];
```

Esta sería la representación gráfica del código:

0	1	2	3	4

1.2 - Dar valor a los elementos de un array

Una vez tenemos nuestro array inicializado, este se crea "vacío" y lo que tenemos que hacer es darle valores. Para ello lo que hacemos es acceder a la posición del array a través del índice, el cual en programación recordamos que los índices empiezan en la posición 0. Y que el último elemento estará colocado en la posición anterior al tamaño del array, en este caso 4 ($n-1$ = última posición)

```
arrayEnteros[0] = 25;  
arrayEnteros[1] = 27;  
arrayEnteros[2] = 25;  
arrayEnteros[3] = 29;  
arrayEnteros[4] = 20;
```

De esta forma asignamos valor a los elementos del array, como vemos accedemos por posición. Esta será la representación en cada una de las líneas de código:

0	1	2	3	4
25				
25	27			
25	27	25		
25	27	25	29	
25	27	25	29	20

Como vimos en el [video sobre las variables y los operadores](#), en este caso también podemos inicializar los arrays cuando los declaramos, y lo hacemos de la siguiente forma.

```
int[] arrayEnteros = { 25, 27, 25, 29, 20 }
```

Lo cual nos genera exactamente el mismo resultado que el ejemplo anterior. Pero como podemos observar no indicamos el tamaño de forma explícita, sino que lo obtiene del número de parámetros que indicamos.

1.3 - Trabajar con el array

Como hemos indicado anteriormente los arrays son inmutables, lo que quiere decir que no los podemos modificar, pero ello no nos impide poder trabajar con ellos.

Normalmente cuando tenemos una colección, ya sean listas o arrays, solemos trabajar con los datos tanto en conjunto como individualmente. Para trabajar individualmente lo que tenemos que hacer es acceder a cada uno de los elementos del array de forma individual, y ello lo haremos a través del índice.

Comúnmente accedemos a los registros uno por uno utilizando un **bucle for** de la siguiente manera:

```
for(int i = 0; i < arrayEnteros.Length; i++){  
    Console.WriteLine($"El numero es: {arrayEnteros[i]}")  
}
```

Como podemos observar en el bucle para indicar el segundo parámetro y saber cuantas iteraciones ejecutar hemos utilizado una función que nos viene dentro de Array, la cual es `Array.Length`.

Como `Array.Length` el propio tipo nos viene con una gran variedad de métodos que podemos utilizar en caso de que los necesitemos.

- `Array.Contains(elemento)` Devolverá `True` o `False` en caso de que el elemento esté dentro del array.
- `Array.Reverse()` Devolverá el array de forma inversa.

Como estos tenemos otros muchos métodos que iremos viendo más adelante conforme trabajemos.

2 - Array multidimensional

No siempre un array común va a cumplir con todo lo que necesitamos, para algunos escenarios necesitaremos un array de n dimensiones. y para ello tenemos los arrays multidimensionales.

2.1 - Declaración del array multidimensional

Muy similar al anterior, únicamente que en este caso en vez de crearnos una lista única nos creará una especie de tabla. Como en el caso anterior, para los arrays multidimensionales, también tenemos que introducir la cantidad de elementos que va a contener.

```
string[,] ciudades= new string[2, 3];
```

Como podemos observar el resultado gráfico sería algo así:

	0	1	2
0			
1			

2.2 - Dar valor a los elementos de un array multidimensional

Para acceder a los elementos del array multidimensional lo hacemos exactamente de la misma forma que accedemos en el array normal. La única diferencia es que en este caso, necesitamos dos índices.

```
ciudades[0, 0] = "Teruel";  
ciudades[0, 1] = "Huesca";  
ciudades[0, 2] = "Zaragoza";  
ciudades[1, 0] = "Valencia";  
ciudades[1, 1] = "castellon";  
ciudades[1, 2] = "Alicante";
```

Y el resultado final de introducir los datos en este array multidimensional

	0	1	2
0	Teruel		
1			
0	Teruel	Huesca	
1			
0	Teruel	Huesca	Zaragoza
1			
0	Teruel	Huesca	Zaragoza
1	Valencia		
0	Teruel	Huesca	Zaragoza
1	Valencia	Castellón	
0	Teruel	Huesca	Zaragoza
1	Valencia	Castellón	Alicante

2.3 - Trabajar con un array multidimensional

El tener que Acceder por dos índices genera que la forma de recorrer el array sea algo diferente. En este caso, necesitaremos bucle for dentro de otro bucle for, para posteriormente acceder con ambos índices `ciudades[i, j]` .

```
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
    {
        Console.WriteLine($"La ciudad es {ciudades[i,j]}");
    }
}
```

3 - Jagged array

Como hemos indicado en el caso anterior, muchas veces necesitamos un array con varias dimensiones. pero tenemos que necesitar el mismo número de dimensiones para todos los elementos del array.

Para poner un ejemplo sencillo, imaginemos que queremos leer de una base de datos todos los cambios de salario de dos empleados. Para ello utilizaremos un array multidimensional, la primera de las filas del array para el primer empleado y la segunda para el segundo.

Para este ejemplo el empleado1 ha cambiado de sueldo 2 veces desde que está trabajando, mientras que el empleado2 ha cambiado de sueldo 7 veces.

Utilizando un array multidimensional, tenemos que crear un array de la siguiente manera `decimal[,] arraySueldos = new decimal[2, 7]` . Este ejemplo, llevado a una lista con 10 mil o 10 millones de registros nos puede causar unos problemas de rendimiento y memoria terribles, ya que para el primer empleado, estamos reservando 4 espacios de memoria que nunca vamos a utilizar. Para evitar este problema, disponemos de los jagged arrays.

3.1 - Declaración de un jagged array

Los jagged array se definen de una forma un tanto diferente, ya que lo que hacemos es mezclar los dos vistos previamente, el array y el array multidimensional.

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

De esta forma nos creará espacio para 3 arrays, cada uno independiente, a los que tenemos que asignarles un tamaño. De la siguiente manera:

```
provincias[0] = new string[3];
provincias[1] = new string[2];
provincias[2] = new string[4];
```

Como podemos observar accedemos por índice a cada uno de los elementos del array e instanciamos un nuevo array dentro del mismo. Este nuevo array es que nos indicará el tamaño del mismo.

La representación gráfica es la siguiente:

	0	1	2	3
0				
1				
2				

Como podemos observar hay huecos en "blanco", eso es porque ese espacio de memoria no esta utilizado.

3.2 - Dar valor a los elementos de un jagged array

Para asignar valores debemos hacer realizar el mismo proceso, que es acceder por índice, la diferencia es que en este caso los índices no son fijos, por lo que a la hora de leer lo haremos de una forma diferente, para asignar valores, lo hacemos directamente, ya que conocemos, o debemos, el tamaño.

```
provincias[0][0] = "Huesca";  
provincias[0][1] = "Zaragoza";  
provincias[0][2] = "Teruel";  
  
provincias[1][0] = "Cáceres";  
provincias[1][1] = "Badajoz";  
  
provincias[2][0] = "A Coruña";  
provincias[2][1] = "Pontevedra";  
provincias[2][2] = "Ourense";  
provincias[2][3] = "Lugo";
```

Y la representación seria la siguiente:

	0	1	2	3
0	Huesca	Zaragoza	Teruel	
1	Cáceres	Badajoz		
2	A Coruña	Pontevedra	Ourense	Lugo

3.3 - Trabajar con un jagged array

En este caso el array es totalmente dinámico, para cada uno de los registros tenemos un tamaño diferente, por lo que para recorrerlo debemos consultar el tamaño de cada uno de los registros, como hemos visto antes con `.Length`.

```
for (int i = 0; i < provincias.Length; i++)
{
    System.Console.WriteLine($"registro ({i}): ", i);

    for (int j = 0; j < provincias[i].Length; j++)
    {
        System.Console.WriteLine("{0}{1}", provincias[i][j], j == (provincias[i].Length
    }
    System.Console.WriteLine();
}
```

```
//Resultado
registro (1):
Hueca Zaragoza Teruel
registro (2):
Cáceres Badajoz
registro (3):
A coruña Pontevedra, ourense Lugo
```

Como nota final indicar que podemos mezclar los arrays multidimensionales con los jagged array, pero sinceramente no lo recomiendo, hay formas mejores de llegar a una solución, por no hablar de que apenas se ven en el mundo laboral.

4 - Listas o List<T>

Ahora dejamos atrás los arrays para pasar a uno de los elementos más utilizados en el entorno C#, las listas o `List<T>` y es un tipo de dato utilizado tan asiduamente gracias a su facilidad y a su gran funcionalidad.

`List<T>` es un tipo de dato, similar al tipo `array`, que acabamos de ver, pero que en este caso son mutables, lo que quiere decir que podemos añadir o quitar elementos cuando queramos.

Otro dato muy importante a tener en cuenta sobre las listas es que implementan `IEnumerable<T>` la cual es una interfaz que permite ejecutar consultas `LINQ`, las cuales veremos en este mismo post. Debido a ello el uso de las listas es continuo en nuestras aplicaciones.

4.1 - Creación de una lista

Cuando creamos una lista a diferencia de los arrays no utilizamos un índice, ya que estas al ser mutables no tienen un tamaño predefinido desde la inicialización. Para replicar el ejemplo

anterior, para crear una lista de string lo hacemos de la siguiente forma:

```
List<string> provincias = new List<string>();
```

Como vemos la `T` de `List<T>` es el tipo de dato que queremos utilizar, y otra vez, este tipo puede ser tanto primitivo como no primitivo.

4.2 - Añadir elementos a una lista

Como hemos indicado anteriormente, ya no accedemos por índice, por lo tanto para añadir elementos lo hacemos de una forma mas dinámica con el metodo `.Add(T)` en el cual debemos pasar por parámetro un tipo de dato del cual es la lista, en nuestro caso `string`.

```
provincias.Add("Teruel");
```

además de esto, si tenemos una lista y queremos añadirla dentro de otra lista, por ejemplo, estamos listando todas las provincias y primero las listamos por comunidades, podemos añadir a una lista, otra lista, utilizando `.AddRange(List<T>)`

```
aragon.Add("Huesca");  
aragon.Add("Zaragoza");  
aragon.Add("Teruel");  
  
provincias.AddRange(aragon);
```

Como observamos en el ejemplo, pasamos por parámetro la lista de aragon.

4.3 - Trabajar con una lista

En este caso, para trabajar con una lista podemos utilizar cualquiera dentro de la infinidad de métodos que el compilador nos provee, aquí vamos a ver sólo unos pocos a modo de ejemplo.

Primero vamos a ver cómo iterar la lista, para ello utilizaremos un bucle `foreach`

```
foreach (string provincia in provincias){  
    Console.WriteLine($"la provincia es {provincia}");  
}
```

Más métodos que podemos utilizar:

- `lista.ToArray()` ; nos convierte la lista en un array.
- `lista.Count` ; nos devuelve un entero con el número de elementos dentro de una lista.
- `lista.First()` ; nos devuelve el primer elemento.
- `lista.Last()` ; nos devuelve el último elemento.
- `lista.Clear()` ; vacía o remueve todos los elementos de la lista.

5 - LINQ

Primero de todo indicar que lo que vamos a ver es una pequeña introducción al lenguaje `LINQ` ya que verlo todo completamente nos da para hacer un centenar de videos.

5.1 - Qué es LINQ

LINQ es un lenguaje de consultas a bases de datos, o al menos esa era su intención original. Mientras LINQ se estaba desarrollando se dieron cuenta que una `List<T>` no deja de ser los resultados que obtenemos sobre hacer una consulta sobre una BBDD así que porqué no aprovecharlo y poder utilizarlo dentro de nuestro código, no solo para consultar, sino, para filtrar.

5.2 - Cómo utilizar LINQ

Utilizar LINQ dentro de nuestro código es muy sencillo, tan solo tenemos que utilizar un tipo que extienda de la clase `IEnumerable<T>`, como puede ser `LIST<T>`.

Para utilizarlo indicaremos: lista `.Where(condicion)`

Dentro de "condicion" debemos indicar el elemntos en el que estamos, comunmente yo utilizo la letra a pero mucha gente utiliza la inicial del elemento que estamos iterando, en este caso (ejemplo provincias) la p.

En nuestro ejemplo `provincias.Where(p => p)` con `p=>p` indicamos donde se va a ejecutar esa consulta, que a su vez, pasa individualmente por cada uno de los elementos de la lista.

Para añadir condiciones a la consulta podemos utilizar las mismas condiciones que disponemos en sus tipos, en este ejemplo para el tipo string utilizaremos el método `.Contains()` que nos devuelve verdadero o falso si el elemento a comprobar contiene el valor pasado por parámetro.

Posteriormente recorreremos el resultado imprimiendo cada uno de los registros.

```
provincias.Add("Huesca");
provincias.Add("Zaragoza");
provincias.Add("Teruel");

IEnumerable<string> provinciasFiltradas = provincias.Where(p => p.Contains("e"));
foreach(string provincia in provinciasFiltradas){
    Console.WriteLine(provincia);
}
```

En este escenario imprime tanto Huesca como Teruel, ya que ambas contienen la letra e (minúscula).

Trabajando con Fechas - DateTime

En este tutorial veremos cómo trabajar con el tipo `Datetime`, el cual nos permite trabajar con fechas. En el mundo laboral, que es básicamente lo importante, utilizamos fechas continuamente, por lo que es importante tener los siguientes conceptos claros.

Índice

- 1 - Creación del objeto `Datetime`
 - 1.1 - Localización con `cultureinfo`
 - 1.2 - Propiedades del tipo `Datetime`
 - 1.3 - Métodos del tipo `Datetime`
 - 1.4 - El tipo `TimeSpan`
- 2 - Comparación de fechas
- 3 - Imprimir la fecha con formato

1 - Creación del objeto `Datetime`

Para crear el objeto `Datetime` tenemos multitud de opciones como podemos observar:

```
public DateTime(long ticks);
public DateTime(long ticks, DateTimeKind kind);
public DateTime(int year, int month, int day);
public DateTime(int year, int month, int day, Calendar calendar);
public DateTime(int year, int month, int day, int hour, int minute, int second);
public DateTime(int year, int month, int day, int hour, int minute, int second, DateTimeKind kind);
public DateTime(int year, int month, int day, int hour, int minute, int second, DateTimeKind kind, Calendar calendar);
public DateTime(int year, int month, int day, int hour, int minute, int second, DateTimeKind kind, Calendar calendar, bool isLeapYear);
public DateTime(int year, int month, int day, int hour, int minute, int second, DateTimeKind kind, Calendar calendar, bool isLeapYear, bool isDaylightSavingTime);
public DateTime(int year, int month, int day, int hour, int minute, int second, DateTimeKind kind, Calendar calendar, bool isLeapYear, bool isDaylightSavingTime, bool isUniversalTime);
public DateTime(int year, int month, int day, int hour, int minute, int second, DateTimeKind kind, Calendar calendar, bool isLeapYear, bool isDaylightSavingTime, bool isUniversalTime, bool isCurrentCulture);
```

Sin embargo dos de ellas son las más comunes, ya que son las que utilizamos cuando queremos comparar fechas, y cuando queremos comparar tiempo además de fechas.

```
public DateTime(int year, int month, int day);
public DateTime(int year, int month, int day, int hour, int minute, int second);
```

Por lo que para crear un objeto de fecha utilizaremos la siguiente sentencia:

```
DateTime testFecha = new DateTime(1989, 11, 2, 11, 15, 16);
```

Donde:

- 1989 es el año.
- 11 es el mes.
- 2 es el día.
- 11 es la hora, en formato 24h, por lo tanto, las 11 de la mañana.

- 15 son los minutos.
- 16 son los segundos.

Otra opción para crear las fechas es utilizar los "ticks" o golpes de reloj, estos cuentan desde el 1 de enero de 1970.

```
public DateTime(long ticks);
```

Porque eligieron el 1 de enero de 1970. Bueno el motivo es que cuando estaban desarrollando unix necesitaban una fecha para empezar a contar y seleccionaron esa. No tiene ningún misterio ni ningún otro motivo oculto.

Finalmente, podemos convertir fechas desde un string, para el cual tenemos dos opciones.

```
DateTime ejemploFecha = Convert.ToDateTime("10/22/2015 12:10:15 PM");
DateTime ejemploFecha = DateTime.Parse("10/22/2015 12:10:15 PM");
```

Ambas opciones son completamente válidas aunque tienen pequeñas diferencias internamente, las dos convertirán el texto en una fecha.

Desafortunadamente el ejemplo que tenemos encima no funciona correctamente, tan solo si estamos en una máquina de estados unidos, donde el formato para las fechas es mes/dia; Si por el contrario estamos en una máquina configurada para europa, el formato correcto para una fecha es dia/mes. *Desconozco Sudamérica.

1.1 - Localización con cultureinfo

Para evitar este problema tenemos el tipo `CultureInfo` que se encuentra dentro de `System.Globalization` el cual nos permite especificar el formato de una fecha, basandonos en el país junto al idioma. El siguiente código nos generaría el tipo `CultureInfo` con español de España.

```
CultureInfo cultureInfoES = new CultureInfo("es-SP");
```

Mientras que si lo que queremos es un país de Sudamérica como por ejemplo Argentina, sería el siguiente código

```
CultureInfo cultureInfoAR = new CultureInfo("es-AR");
```

Por lo que para el ejemplo visto anteriormente (mes/dia) tenemos que utilizar el `CultureInfo` de Estados Unidos:

```
CultureInfo cultureInfoUS = new CultureInfo("en-us");
DateTime ejemploFecha = Convert.ToDateTime("10/22/2015 12:10:15 PM", cultureInfoUS)
```

Nota muy importante: Si indicamos mal el `CultureInfo`, el programa crasheará y detendrá su ejecución así que hay que estar seguros con lo que hacemos.

1.2 - Propiedades del tipo Datetime

A raíz de la fecha anterior podemos obtener la mayoría de la información que comúnmente utilizamos, como pueden ser día, mes, día de la semana, et. Todo ello nos viene dentro del Tipo Datetime, por lo que no tenemos que crear ningún método para obtenerlo.

Aquí muestro una pequeña lista con algunos ejemplos:

```
int dia = fecha.Day; //Nos devuelve el día del mes en número (1, 2...30, 31)
int mes = fecha.Month; //Nos devuelve el número de mes
int year = fecha.Year; //Nos devuelve el año
int hora = fecha.Hour; //Devuelve la hora
int minuto = fecha.Minute; //devuelve el minuto
int segundo = fecha.Second; //devuelve los segundos.
string diaDeLaSemana = fecha.DayOfWeek; //Devuelve el día de la semana con letra (D)
int diaDelyear = fecha.DayOfYear; //Devuelve en que día del año estamos, con número
DateTime tiempo= fecha.Date; //devuelve horas:minutos:segundos
```

1.3 - Métodos del tipo Datetime

como en el caso anterior, el propio tipo nos trae infinidad de métodos para ser ejecutados. entre los que podemos añadir días, meses o incluso tiempo.

```
fecha.AddDays(1); //añadira un día a la fecha actual
fecha.AddMonths(1); //Añadira un mes a la fecha actual.
fecha.AddYears(1); //Añadira un año a la fecha actual.
```

Como vemos todos los ejemplos son de añadir, pero qué pasa, si lo que queremos es restar días, para ello también tenemos que utilizar el método de añadir, solo que el valor que le pasamos será negativo.

```
fecha.AddDays(-1); //restará un día a la fecha actual
fecha.AddMonths(-1); // restará un mes a la fecha actual.
fecha.AddYears(-1); //restará un año a la fecha actual.
```

Por último si lo que queremos es añadir tiempo, tenemos que utilizar un tipo concreto.

1.4 - El tipo TimeSpan

Similar al caso anterior, solo que ahora en el constructor podemos tener las opciones del tiempo

```
public TimeSpan(long ticks);
public TimeSpan(int hours, int minutes, int seconds);
public TimeSpan(int days, int hours, int minutes, int seconds);
public TimeSpan(int days, int hours, int minutes, int seconds, int milliseconds);
```

Como observamos, para crear el tiempo `TimeSpan` podemos hacerlo desde días, hasta los milisegundos. y lo realizaremos de la siguiente forma:

```
DateTime fecha = new Datetime(2019, 01, 01);
//fecha tendrá el valor de 1 de enero de 2019 a las 00h 00m

TimeSpan tiempo = new TimeSpan (1, 5, 30, 5);
//creamos un objeto timespan con valor de 1 dia, 5 horas, 30 minutos, 5 segundos

DateTime fechaActualizada = fecha.Add(tiempo);
//Sumamos el tiempo a la fecha anterior
//Resultado: 2 de enero de 2019 a las 5:30 de la mañana.
```

2 - Comparación de fechas

Comparar fechas es muy importante en el mundo real y es algo que hay que tener muy claro. Por ejemplo comparamos fechas dentro de filtros o en consultas a la base de datos, o consultas **LINQ** como la que vimos en el post anterior.

Cuando comparamos lo podemos hacer de dos formas:

- Utilizar el método `DateTime.Compare(fecha1, fecha2)` dentro del tipo estático `DateTime`.
- Utilizar la propia fecha para compararla con la segunda, `fecha1.CompareTo(fecha2)`;

En ambos casos el resultado que nos devuelve es el mismo, nos devolverá un entero (int) que corresponde con lo siguiente:

- **menor que 0** si la primera fecha es menor que la segunda.
- **0** si ambas fechas son iguales
- **mayor que 0** si la primera fecha es mayor que la segunda.

```
DateTime fecha1 = new DateTime(1989, 11, 2);
DateTime fecha2 = new DateTime(1978, 4, 15);
int fechaResultado = DateTime.Compare(fecha1, fecha2);
//0 indistintamente
int fechaResultado = fecha1.CompareTo(fecha2);

if (fechaResultado < 0)
{
    Console.WriteLine("La primera fecha es menor");
}
else if (fechaResultado == 0)
{
    Console.WriteLine("Las fechas son iguales");
}
else
{
}
```

```
Console.WriteLine("La segunda fecha es menor");
```

```
}
```

3 - |

Fin: ón
que je
fijar
Por or
defi

```
DateTime fecha= new DateTime(1989, 11, 2, 11, 15, 16);  
fecha.ToString(); // resultado: 02/11/1989 11:15:16  
fecha.ToShortDateString(); //resultado: 02/11/1989  
fecha.ToLongDateString(); //Resultado: Jueves 2 octubre 1989  
fecha.ToShortTimeString(); //resultado: 11:15
```

Además de las opciones que nos trae por defecto, podemos crear nuestra propia versión utilizando el método `.ToString()` ya que si le pasamos un valor por parámetro el compilador lo traduce a lo que necesitamos.

Los ejemplos anteriores se pueden representar tan solo con `.ToString()` de la siguiente manera:

Pero de qué nos sirve poder imprimir algo que ya podemos imprimir anteriormente. El método `.ToString()` es mucho más potente, ya que podemos pasarle una combinación de caracteres para que muestre el formato tal y como lo necesitamos. por ejemplo `yyyy` se traduce al año, `MM` se traduce al mes.

Algo muy común en todas las aplicaciones es tener un log, los logs guardan fecha y hora con precisión de microsegundo. Podemos crear un mensaje partiendo de una fecha que el formato sea similar al de un log.

```
DateTime fecha= new DateTime(1989, 11, 2, 11, 15, 16, 123);  
fecha.ToString(yyy-MM-ddThh:mm:ss.ms);  
// resultado: 1989-01-11T11:15:16.123
```

Como podemos observar los caracteres como `-` o `.` también salen representados en el resultado.

Además de estos, tenemos muchos más códigos para pasar en el método `.ToString()` posteriormente podemos utilizar cualquiera de los métodos disponibles en la clase **string**.

Finalmente una tabla con todo lo que el método `.ToString()` permite añadir, notese la diferencia entre mayúsculas y minúsculas.

Epecificación	Descripción	Salida
d	Fecha corta	02/11/1989
D	Fecha larga	jueves 2 Noviembre 1989
t	Tiempo corto	11:15
T	Tiempo largo	11:16:16
f	Fecha y hora completa corto	Jueves 2 Noviembre 1989 11:15
F	Fecha y hora completa largo	Jueves 2 Noviembre 1989 11:15:16
g/G	fecha y hora por defecto	02/11/1989 11:15
M	Día y mes	02-Nov
r	RFC 1123 fecha	Jue 02 Nov 1989 11:15:16 GMT
s	fecha y hora para ordenar	1989-11-02T11:15:16
u	tiempo universal, con timezone	1989-11-02T11:15:16z
Y	mes año	Noviembre 1989
dd	Día	2
ddd	día corto	Jue
dddd	día completo	Jueves
hh	hora con 2 digitos	11

HH	hora con 2 digitos formato 24h	23
mm	minuto con 2 digitos	15
MM	mes	11
MMM	nombre mes corto	Nov
MMMM	nombre mes largo	Noviembre
ss	segundos	16
fff	milisegundos	123
tt	AM/PM	PM
yy	año con 2 digitos	89
yyyy	año con 4 digitos	1989

Trabajando con Ficheros

Índice

1 - Rutas absolutas y relativas

2 - Leer un fichero

2.1 - Tipo StreamReader

2.2 - Tipo File

3 - Escribir un fichero

3.1 - Tipo StreamWriter

3.2 - Tipo File

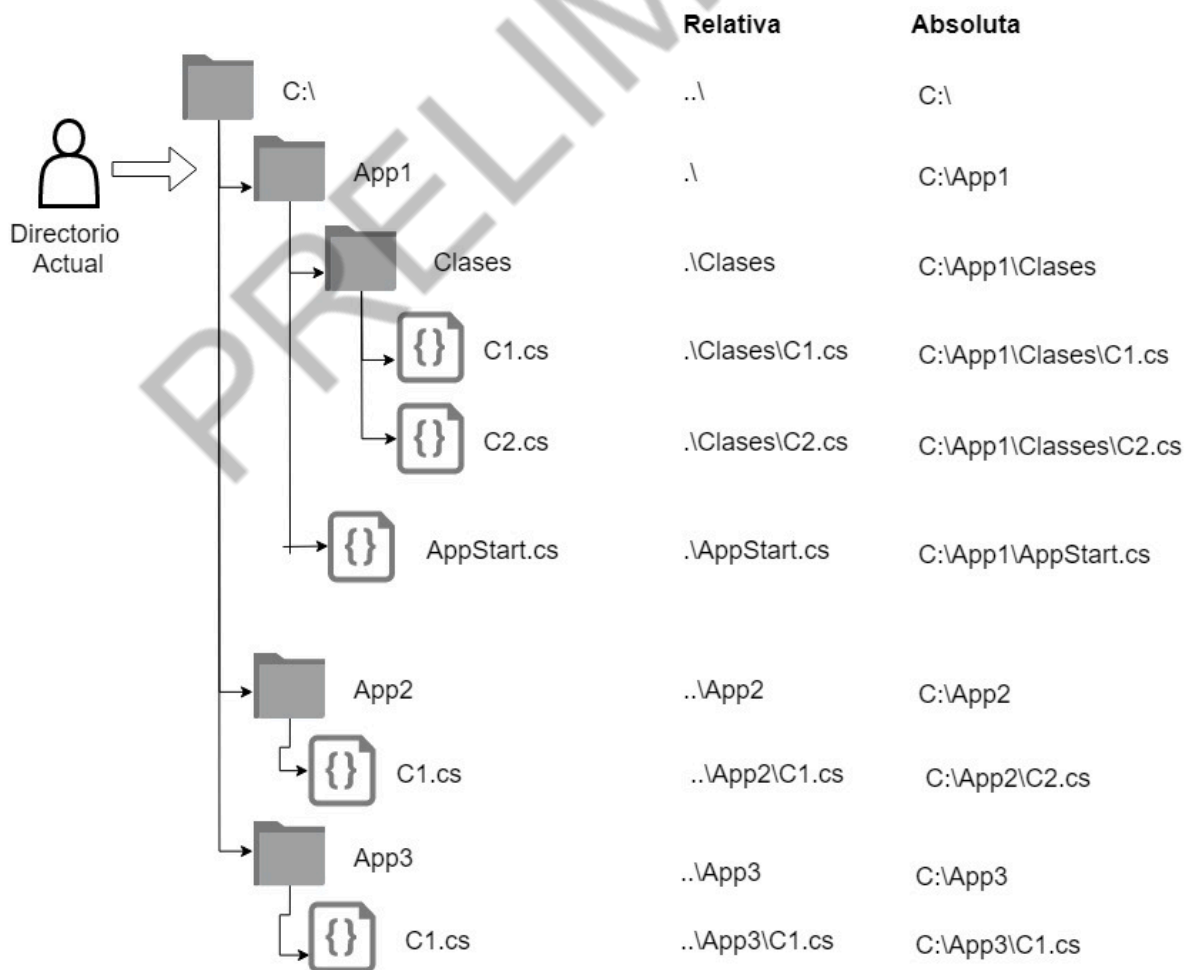
1 - Rutas absolutas y relativas

Lo primero que veremos serán las rutas absolutas y relativas, ya que estas son las que utilizaremos para acceder a los propios ficheros.

La diferencia entre ambas es muy simple:

- En las rutas relativas accedemos en función de la carpeta o directorio en el que nos encontramos.
- Mientras que en las rutas absolutas accedemos basándonos en el directorio raíz del ordenador. Comúnmente `C:\` en Windows o `/` en Linux.

Esto sería un ejemplo gráfico de las rutas relativas y absolutas:



Como vemos a la izquierda nos ubicamos inicialmente en el directorio `C:\App1` por lo que para subir un submenu indicamos `..\` mientras que en las rutas absolutas da igual donde nos coloquemos, ya que siempre inicializamos indicando `C:\`

En el mundo laboral debido a que las aplicaciones pueden estar ubicadas en distintos servidores se utilizan rutas absolutas para ubicar los ficheros.

2 - Leer un fichero

2.1 - Tipo StreamReader

Para leer un fichero ya existente, utilizamos el tipo `StreamReader()` y este a su vez lo pondremos dentro de una sentencia `using`.

Por qué lo instanciamos dentro de la sentencia `using` ?

Porque cuando el bloque de código que crea el `using` se cierra, automáticamente destruye de la memoria todas las variables inicializadas dentro de él. Esto obviamente no es problema para una aplicación muy pequeña, pero cuando tenemos una app muy grande, o que requiere leer varios GB de información, puede llegar a ser un problema muy grande, y hacer fallar nuestro sistema.

`StreamReader` nos trae por defecto un método llamado `.ReadLine()` el cual lee cada línea del documento, y podremos trabajar con él.

```
//Accedemos al fichero
using (StreamReader reader = new StreamReader("c:\ejemplo\fichero.txt"))
{
    string text; //Creamos la variable que contendrá el texto
    while ((text = reader.ReadLine()) != null) //Leemos línea por línea
    {
        Console.WriteLine(text); //Mostramos la información
    }
}
```

2.2 - Tipo File

Otra opción es la clase `File`, la cual nos leerá el fichero de un solo volcado, como se observa en todos mis posts, me preocupa en cierta medida la memoria, así que si el fichero es muy grande quizá no sea la mejor idea.

```
string filereader = File.ReadAllText("c:\ejemplo\fichero.txt");
```

3 - Escribir un fichero

3.1 - Tipo StreamWriter

Para escribir el proceso también es muy sencillo, en este caso utilizaremos el tipo `StreamWriter` y además de ello, deberemos indicar el texto o la lista de texto a escribir.

```
string[] paises = new string[] { "USA", "Inglaterra", "Alemaia" };  
using (StreamWriter writer = new StreamWriter("c:\\ejemplo\\fichero.txt"))  
{  
    foreach (string item in countries)  
    {  
        writer.WriteLine(item);  
    }  
}
```

Este método tiene una pequeña pega, y es que, para añadir más adelante una línea de texto, lo que hace por detrás el código es reescribir el fichero entero. Además, si este fichero no existe, nos lo creará.

3.2 - Tipo File

Como en el caso anterior, podemos utilizar el tipo `File` para escribir, en este caso tenemos varias opciones

```
string[] paises = new string[] { "USA", "England", "Germany" };  
  
File.WriteAllText("c:\\ejemplo\\fichero.txt", "China Japon Korea");  
File.WriteAllLines("c:\\ejemplo\\fichero.txt", paises);
```

- `File.WriteAllText()` escribirá todo el texto de un solo volcado.
- `File.WriteAllLines()` escribirá línea por línea cada uno de los elementos que mandamos en el `array`.

Programación orientada a objetos y clases

En este tutorial veremos cómo implementamos la realidad a nuestras aplicaciones, para ello utilizaremos la programación orientada a objetos, también llamada POO.

Índice

- 1 - Qué es la programación orientad a objetos?
- 2 - Crear una clase
 - 2.1 - Crear una instancia de una clase
- 3 - Constructores
- 4 - Herencia

1 - Qué es la programación orientad a objetos?

Es un paradigma que convierte objetos reales a entidades de código llamadas clases. Esto puede parecer muy complejo, pero no lo es para nada.

Imaginémonos un objeto del mundo real como puede ser una moto



Como podemos observar en la imagen, disponemos de dos entidades. **Piloto** y **Moto** .

Dependerá de la lógica de negocio saber si es la moto la que contiene al piloto, o es el piloto el que contiene la moto. Para nuestro ejemplo, la moto contendrá al piloto.

Cuando convertimos la moto a un objeto, debemos pensar en sus características, como marca, modelo, número de ruedas, cilindrada, etc. Estas características serán nuestras propiedades.

Además de eso, en la moto realizas acciones, como son arrancar, acelerar, frenar, cambiar de marcha, todas estas acciones serán los métodos.

Por lo tanto, una clase es un tipo específico que sirve para crear objetos.

Como recordamos de tutoriales anteriores, tenemos tipos primitivos como `int` , `decimal` , etc. Las clases son un tipo más, pero personalizado a nuestro gusto.

2 - Crear una clase

Para crear una clase debemos indicar la palabra registrada `class` y posteriormente el nombre que queremos utilizar. Como en todos los bloques de código, definiremos cuando empieza y cuando acaba utilizando las llaves `{` y `}` , dentro del bloque de código, introduciremos nuestras propiedades y nuestros métodos.

```
class Moto
{
    public decimal VelocidadMaxima { get; set; }
    public int NumeroRuedas { get; set; }
    public Motorista Piloto { get; set; }

    public Acelerar(){
        //Código aquí
    }
    public Arrancar(){
        //Código aquí
    }
}
```

Como vemos además utilizamos la palabra reservada `public` la cual es un modificador de acceso, que podemos ver en el curso de los modificadores de acceso.

2.1 - Crear una instancia de una clase

Un objeto es la representación en memoria de una clase y puedes crear tantos objetos de una clase como quieras, para ello utilizaremos la palabra reservada `new` .

```
Moto aprilia = new Moto();
```

Posteriormente le podemos dar valores a sus propiedades, así como llamar a sus métodos. Por supuesto, al disponer de `Piloto` dentro del tipo `Moto` , también le podemos dar valores

```
aprilia.VelocidadMaxima = 320;
aprilia.NumeroRuedas = 2;
aprilia.Acelerar();

aprilia.Piloto.Nacionalidad = "ESP";
```

3 - Constructores

Como hemos observado, cuando creamos la instancia de la clase utilizamos `new Moto()` bien, esta sentencia lo que hace es llamar al constructor por defecto dentro de `System.Object`, pero claro, que pasa si no queremos utilizar el constructor por defecto, ya que el constructor por defecto nos inicializa todas las variables a null.

Para ello podemos crear un constructor personalizado por nosotros mismos, donde podremos asignar valores manualmente, como podemos observar en el ejemplo. Ahora cuando realizamos `Moto aprilia = new Moto();` nos creara por defecto una `VelocidadMaxima` de 320 y un `NumeroRuedas` de 2;

```
class Moto
{
    public decimal VelocidadMaxima { get; set; }
    public int NumeroRuedas { get; set; }
    public Motorista Piloto { get; set; }
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public Moto(){
        VelocidadMaxima = 320;
        NumeroRuedas = 2;
    }

    public Acelerar(){
        //Código aquí
    }
    public Arrancar(){
        //Código aquí
    }
}
```

Pero ¿qué pasa si tenemos múltiples motos?

En nuestro ejemplo al tener los valores asignados en el constructor por defecto se nos crearían todas las motos con los mismos valores, esto quiere decir que en las siguientes sentencias

```
Moto aprilia = new Moto();
Moto Ducati = new Moto();
```

Ambas motos tendrán una `VelocidadMaxima` de 320 y un valor de 2 en `NumeroRuedas` , por lo que los valores podrían no ser correctos.

Para arreglar este problema, lo que hacemos es añadir un constructor que acepte valores por parámetro. Utilizando Sobrecarga de operadores podemos crear tantos constructores como deseemos.

```
class Moto
{
    public decimal VelocidadMaxima { get; set; }
    public int NumeroRuedas { get; set; }
    public Motorista Piloto { get; set; }
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public Moto(decimal velocidadMaxima, int NumeroRuedas){
        VelocidadMaxima = velocidadMaxima;
        NumeroRuedas = NumeroRuedas;
    }

    public Moto(string marca, string modelo){
        Marca = marca;
        modelo = modelo;
    }

    public Moto(){
        VelocidadMaxima = 320;
        NumeroRuedas = 2;
    }

    public Acelerar(){
        //Código aquí
    }

    public Arrancar(){
        //Código aquí
    }
}
```

Como podemos comprobar, ambos constructores funcionan a la perfección

```
Moto aprilia = new Moto("Aprilia", "RX");
Moto motoSinMarca = new Moto(210,2);
```


Otra característica muy importante dentro de la programación orientada a objetos es la herencia. La cual nos permite heredar información de una clase padre a su hijo.

En el ejemplo anterior de la moto, supongamos que tenemos también un coche, ambos son vehículos, por lo que ambos tienen características comunes, como pueden ser marca, modelo, numero de ruedas, y otras diferentes como pueden ser la cilindrada para las motos o la tracción para los coches. El uso de la Herencia nos permite ahorrar multitud de líneas de código y complejidad dentro de nuestros programas.

```
class Vehiculo
{
    public decimal VelocidadMaxima { get; set; }
    public int NumeroRuedas { get; set; }
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public Vehiculo(decimal velocidadMaxima, int NumeroRuedas){
        VelocidadMaxima = velocidadMaxima;
        NumeroRuedas = NumeroRuedas;
    }

    public Vehiculo(string marca, string modelo){
        Marca = marca;
        Modelo = modelo
    }

    public Acelerar(){
        //Código aquí
    }

    public Arrancar(){
        //Código aquí
    }
}

class Moto : Vehiculo
{
    public int Cilindrada { get; set; }

    public Moto(decimal velocidadMaxima, int NumeroRuedas, int cilindrada) : base(
        Cilindrada = cilindrada;
    }

    public Moto(string marca, string modelo, int cilindrada) : base(marca, modelo)
        Cilindrada = cilindrada;
    }
}
```

```
public void HacerCaballito(){
    //codigo
}
}

class Coche : Vehiculo
{
    public string Traccion { get; set; }

    public Coche(string marca, string modelo, string Traccion) : base(marca, modelo)
    {
    }

    public bool CerrarPuertas(){
    }

}
}
```

Como podemos observar tanto `Coche` como `Moto` heredan, utilizando `:`, de la clase `Vehiculo`, esto quiere decir que desde ambas clases podemos acceder a los métodos y propiedades de la clase `Vehiculo`, además de a las suyas propias.

Otro detalle interesante es que como podemos observar en el constructor, desde la clase hijo se llama al constructor de la clase padre utilizando la palabra reservada `: Base()` y mandando los parámetros que ese constructor necesita.

Live coding serpiente

El Post de hoy consiste en un juego muy común que todos conocemos de los antiguos Nokia.

La idea de este ejercicio es ver o trabajar la gran mayoría de elementos y técnicas de programación que hemos estado viendo a lo largo del [curso de programación básica](#).

Índice

- 1 - Análisis
- 2 - Código
 - 2.1 - Entidad tablero
 - 2.2 - Entidad Serpiente
 - 2.3 - Entidad Caramelo
- 3 - Lógica del programa
 - 3.1 - Serpiente se mueve
 - 3.2 - Serpiente come
 - 3.3 - Moverse en diferentes direcciones
 - 3.4 - Dibujar el caramelo
 - 3.5 - Morir
 - 3.6 - Juntar las partes

El Juego de la serpiente o snake

En este post veremos una solución a como podemos realizar este juego en .net

1 - Análisis

Lo primero de todo realizaremos un análisis de los requisitos del juego que son a groso modo:

- Tenemos un tablero o un terreno donde podemos mover la serpiente
- La serpiente se puede mover
 - Arriba
 - Abajo
 - Derecha
 - Izquierda
- Caramelos que salen en la pantalla para comerselos y sumar puntos

Por lo tanto si convertimos las entidades reales a entidades de código, [como vimos en el video de objetos y clases](#), obtendremos lo siguiente:

- . Clase Tablero
 - Propiedades:
 - i. Altura
 - ii. Anchura
 - Métodos:
 - i. Dibujar el tablero
- . Clase serpiente
 - Propiedades:
 - i. Cola = Lista de posiciones
 - ii. Dirección actual
 - iii. Puntos

- Métodos
 - i. Morir
 - ii. Moverse
 - iii. Comerse el caramelo
- Clase Caramelo
- Aparece

Además de estas clases, quizá debemos crear alguna adicional, como por ejemplo una clase `Util` que nos dibuje por pantalla una línea o caracter, ya que es una acción que vamos a repetir constantemente.

```
static class Util
{
    public static void DibujarPosicion(int x, int y, string caracter)
    {
        Console.SetCursorPosition(x, y);
        Console.WriteLine(caracter);
    }
}
```

Como podemos observar el método `Console.SetCursorPosition(x,y)` nos ubica el cursor dentro de la terminal en el punto que deseamos.

Y como vimos en el video de **entrada salida por teclado y pantalla**, `Console.WriteLine(string)` nos escribe un texto, en la gran mayoría de casos en este ejercicio, es tan solo un carácter. En el caso de los muros es el caracter `#` o la `x` para la serpiente

2 - Código

Cuando programamos debemos pensar como si tuviéramos ese objeto, u objetos, delante, por lo que el primer elemento que pondríamos sería el tablero, así que la entidad Tablero será la primera que vayamos a crear.

2.1 - Entidad tablero

Creemos la clase tablero con las propiedades `Altura` y `Anchura`, además, ya que la utilizaremos más adelante, creamos la propiedad `ContieneCaramelo` la cual la inicializamos a `false`, ya que cuando dibujamos el tablero, de primera instancia, el caramelo no existe.

También debemos crear el método `DibujarTablero` el cual recorre la altura y la anchura para dibujar el tablero indicado.

```
public class Tablero
{
```

```

public readonly int Altura;
public readonly int Anchura;
public bool ContieneCaramelo;
public Tablero(int altura, int anchura)
{
    Altura = altura;
    Anchura = anchura;
    ContieneCaramelo = false;
}

public void DibujarTablero()
{
    for (int i = 0; i <= Altura; i++)
    {
        Util.DibujarPosicion(Anchura, i, "#");
        Util.DibujarPosicion(0, i, "#");
    }

    for (int i = 0; i <= Anchura; i++)
    {
        Util.DibujarPosicion(i, 0, "#");
        Util.DibujarPosicion(i, Altura, "#");
    }
}
}

```

2.2 - Entidad Serpiente

Por supuesto es muy importante que creamos la entidad serpiente. Para ello es importante que recordemos los datos anteriores. Como hemos dicho, contendrá una lista de posiciones, por lo que `Posicion` en si misma será una entidad. La cual contendrá la posición sobre el eje de las X y la posición sobre el eje de las Y.

```

public class Posicion
{
    public int X;
    public int Y;

    public Posicion(int x, int y)
    {
        X = x;
        Y = y;
    }
}

```

Además de `List<Posicion>` deberá contener la dirección en la que se va a mover. Estas al ser solamente 4 y ser siempre las mismas, las crearemos en una enumeración

```
public enum Direccion
{
    Arriba,
    Abajo,
    Izquierda,
    Derecha
}
```

Utilizando la enumeración nos resulta mucho mas fácil saber que dirección estamos apuntando.

Finalmente la cla

```
public class Serpiente
{
    List<Posicion> Cola { get; set; }
    public Direccion Direccion { get; set; }
    public int Puntos { get; set; }

    public Serpiente(int x, int y)
    {
        Posicion posicionInicial = new Posicion(x, y);
        Cola = new List<Posicion>() { posicionInicial };
        Direccion = Direccion.Abajo;
        Puntos = 0;
    }

    public void DibujarSerpiente()
    {
        foreach (Posicion posicion in Cola)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Util.DibujarPosicion(posicion.X, posicion.Y, "x");
            Console.ResetColor();
        }
    }

    public bool ComprobarMorir(Tablero tablero)
    {
        throw new NotImplementedException();
    }
}
```

```

public void Moverse(bool haComido)
{
    throw new NotImplementedException();
}

public bool ComeCaramelo(Caramelo caramelo, Tablero tablero)
{
    throw new NotImplementedException();
}
}

```

Fin:
y de
cua
Cor
col:
cód
Mie
El r

ón
la
la
El
a.

2.3
No
nos
la e

lo
os

```

public class Caramelo
{
    public Posicion Posicion { get; set; }

    public Caramelo(int x, int y)
    {
        Posicion = new Posicion(x, y);
    }

    public void DibujarCaramelo()
    {
        Console.ForegroundColor = ConsoleColor.Blue;
        Util.DibujarPosicion(Posicion.X, Posicion.Y, "O");
        Console.ResetColor();
    }

    public static Caramelo CrearCaramelo(Serpiente serpiente, Tablero tablero)

```

```

    {
        throw new NotImplementedException();
    }
}

```

3 - l

Aho
imp
vay

Par
se c

Par
incl
pro
se l

in
y

je

to
la
je

```

static void Main(string[] args)
{
    Tablero tablero = new Tablero(20, 20);

    Serpiente serpiente = new Serpiente(10, 10);
    Caramelo caramelo = new Caramelo(0, 0);
    bool haComido = false;

    do
    {
        Console.Clear();
        tablero.DibujarTablero();
        serpiente.Moverse(haComido);
        haComido = serpiente.ComeCaramelo(caramelo, tablero);
        serpiente.DibujarSerpiente();

    } while (serpiente.ComprobarMorir(tablero));

    Console.ReadKey();
}

```


Debemos inicializar tanto el `tablero` como la `serpiente` como el primer `caramelo` fuera del bucle, ya que si está dentro del mismo estaríamos empezando desde 0 todo el rato.

Además, la primera línea dentro del bucle debe ser para limpiar la consola, utilizando un `Console.Clear()` ya que esta no se limpia automáticamente.

Finalmente, de esta pequeña parte de código, debemos recordar que la serpiente primero se mueve y luego come. Finalizando con dibujarla.

3.1 - Serpiente se mueve

El primer paso del programa es moverse, para ello deberemos modificar la posición en la que nos encontramos, dependiendo de la dirección:

- **Arriba:** Y -1
- **Abajo:** Y+1
- **Derecha:** X +1
- **Izquierda:** X-1

Obtenemos la primera posición de la cola (osea la cabeza) y actualizamos su valor como podemos ver en `ObtenerNuevaPrimeraPosicion()` .

```
public void Moverse(bool haComido)
{
    List<Posicion> nuevaCola = new List<Posicion>();
    nuevaCola.Add(ObtenerNuevaPrimeraPosicion());
    nuevaCola.AddRange(Cola);

    if (!haComido)
    {
        nuevaCola.Remove(nuevaCola.Last());
    }

    Cola = nuevaCola;
}

private Posicion ObtenerNuevaPrimeraPosicion()
{
    int x = Cola.First().X;
    int y = Cola.First().Y;

    switch (Direccion)
    {
        case Direccion.Abajo:
            y += 1;
            break;
        case Direccion.Arriba:
            y -= 1;
    }
}
```

```

        break;
    case Direccion.Derecha:
        x += 1;
        break;
    case Direccion.Izquierda:
        x -= 1;
        break;
    }
    return new Posicion(x, y);
}

```

Par
val

Ent
list

Por

Po
10
10
10

El
sigu

Po
10

Par

Po
10
10
10
10

Per
el t
con

Cor

Posición X	Posición Y
10	9
10	10
10	11

Como observamos son las nuevas posiciones de la serpiente.

3.2 - Serpiente come

Para comprobar si la serpiente ha comido, debemos comprobar que el caramelo no esta en ninguna posición de la cola. No debería pasar que este se cree en una posición de la cola, pero por si acaso las comprobamos todas.

Dentro de la clase serpiente el código para comprobar si hemos comido es el siguiente:

```
public bool ComeCaramelo(Caramelo caramelo, Tablero tablero)
{
    if (PosicionEnCola(caramelo.Posicion.X, caramelo.Posicion.Y))
    {
        Puntos += 10; // sumamos puntos
        tablero.ContieneCaramelo = false; //Quitar el caramelo o generar uno nuevo
        return true;
    }
    return false;
}

public bool PosicionEnCola(int x, int y)
{
    return Cola.Any(a => a.X == x && a.Y == y);
}
```

`ComeCaramelo` , devuelve `true` si nos comemos el caramelo, además indica al tablero que ya no tiene un caramelo. Lo que obligará al mismo a generar uno nuevo.

3.3 - Moverse en diferentes direcciones

Pero claro, tal y como está ahora solo nos movemos en una dirección. Lo que debemos hacer es permitir que el usuario introduzca la dirección para la que queremos que se mueva.

Para ello dentro de nuestro bucle debemos introducir el siguiente código:

```
var sw = Stopwatch.StartNew();
while (sw.ElapsedMilliseconds <= 250)
{
    serpiente.Direccion = Util.LeerMovimiento(serpiente.Direccion);
}
```

El método `LeerMovimiento` nos devolvera la `Direccion` que hemos pulsado, si no pulsamos una nueva dirección durante 200milisegundos, devolvera automaticamente la dirección actual.

```
static Direccion LeerMovimiento(Direccion movimientoActual)
{
    if (Console.KeyAvailable)
    {
        var key = Console.ReadKey().Key;
```

```

        if (key == ConsoleKey.UpArrow && movimientoActual != Direccion.Abajo)
        {
            return Direccion.Arriba;
        }
        else if (key == ConsoleKey.DownArrow && movimientoActual != Direccion.Arriba)
        {
            return Direccion.Abajo;
        }
        else if (key == ConsoleKey.LeftArrow && movimientoActual != Direccion.Derecha)
        {
            return Direccion.Izquierda;
        }
        else if (key == ConsoleKey.RightArrow && movimientoActual != Direccion.Izquierda)
        {
            return Direccion.Derecha;
        }
    }
    return movimientoActual;
}

```

Debemos evitar ir en dirección contraria a la que vamos actualmente, o eso causará que choquemos y muramos contra nosotros mismos.

3.4 - Dibujar el caramelo

Dibujar el caramelo es muy sencillo, realizaremos una acción similar a cuando dibujamos la serpiente o los muros. La única diferencia es que, para dibujar el caramelo, debemos estar seguros de dos factores.

- Es totalmente aleatorio
- No se encuentra en los muros ni en la serpiente.

```

public void DibujarCaramelo()
{
    Console.ForegroundColor = ConsoleColor.Blue;
    Util.DibujarPosicion(Posicion.X, Posicion.Y, "O");
    Console.ResetColor();
}

public static Caramelo CrearCaramelo(Serpiente serpiente, Tablero tablero)
{
    bool carameloValido;
    int x,y;

    do

```

```

{
    Random random = new Random();
    x = random.Next(1, tablero.Anchura - 1);
    y = random.Next(1, tablero.Altura - 1);
    carameloValido = serpiente.PosicionEnCola(x, y);

} while (carameloValido);

tablero.ContieneCaramelo = true;
return new Caramelo(x, y);
}

```

3.5

Par

- La
- La

```

public void ComprobarMorir(Tablero tablero)
{
    //Si nos chocamos contra nosotros
    Posicion primeraPosicion = Cola.First();

    EstaViva = !((Cola.Count(a => a.X == primeraPosicion.X && a.Y == primeraPosicion.Y) > 1) || CabezaEstaEnPared(tablero, Cola.First()));
}

//si la primera posicion esta en cualquiera de los muros, morimos.
private bool CabezaEstaEnPared(Tablero tablero, Posicion primeraPosicion)
{
    return primeraPosicion.Y == 0 || primeraPosicion.Y == tablero.Altura
        || primeraPosicion.X == 0 || primeraPosicion.X == tablero.Anchura;
}

```

3.6 - Juntar las partes

Finalmente, para que todo funcione correctamente debemos juntar cada una de las partes, en un orden con sentido. Para ello dentro de la clase `main` dividiremos el código de la siguiente manera.

Primero, como hemos visto antes inicializamos las variables que necesitamos, ellas incluye, el `tablero`, la `serpiente`, el `caramelo` y la variable `haComido`.

```
Tablero tablero = new Tablero(20, 20);

Serpiente serpiente = new Serpiente(10, 10);
Caramelo caramelo = new Caramelo(0, 0);
bool haComido = false;
```

Una vez tenemos todo inicializado, debemos pasar a dibujarlo, recordamos que esta todo en un bucle `do While` . Dentro del bucle, debemos primero de todo, limpiar la pantalla, y después dibujarlo todo. siempre y cuando la serpiente este viva.

Podemos hacer la comprobación de si esta viva tanto al principio del búcle, como al final del mismo. no importa. pero si lo hacemos al principio, debemos ejecutar los movimientos, creacion de la serpiente y el caramelo únicamente si seguimos vivos. En caso opuesto, mostraremos el mensaje de que hemos muerto y una puntuación.

Siendo el siguiente código el resultado de la clase main

```
static void Main(string[] args)
{
    Tablero tablero = new Tablero(20, 20);

    Serpiente serpiente = new Serpiente(10, 10);
    Caramelo caramelo = new Caramelo(0, 0);
    bool haComido = false;

    do
    {
        Console.Clear();
        tablero.DibujarTablero();
        //movemos y comprobamos si ha comido en el turno anterior.
        serpiente.Moverse(haComido);

        //Comprobamos si se ha comido el caramelo
        haComido = serpiente.ComeCaramelo(caramelo, tablero);

        //Dibujamos serpiente
        serpiente.DibujarSerpiente();

        //Si no contiene el caramelo, instanciamos uno nuevo.
        if (!tablero.ContieneCaramelo)
        {
            caramelo = Caramelo.CrearCaramelo(serpiente, tablero);
        }

        //Dibujamos caramelo
```

```
caramelo.DibujarCaramelo();

//Leemos informacion por teclado de la direccion.
var sw = Stopwatch.StartNew();
while (sw.ElapsedMilliseconds <= 250)
{
    serpiente.Direccion = Util.LeerMovimiento(serpiente.Direccion);
}

} while (serpiente.ComprobarMorir(tablero));

Util.MostrarPuntuación(tablero, serpiente);

Console.ReadKey();
```

```
}
```

Poc

PRELIMINAR

Modificadores de acceso

En este tutorial veremos todo lo referente a los modificadores de acceso.

Índice

- 1 - Qué es un modificador de acceso
- 2 - Modificadores de acceso
 - public
 - Private
 - internal
 - protected
 - protected internal
 - private protected

1 - Qué es un modificador de acceso

Un modificador de acceso es aquella cláusula de código que nos indica si podemos acceder o no a un bloque de código específico desde otra parte del programa.

Hay una gran variedad de modificadores de acceso, y estos son aplicados tanto a métodos, propiedades, o clases.

2 - Modificadores de acceso

public

Acceso no restringido que permite acceder a sus miembros desde cualquier parte del código al que se le hace referencia.

```
public class EjemploPublic
{
    public string PruebaAcceso { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        EjemploPublic ejemplo = new EjemploPublic();
        Console.WriteLine(ejemplo.PruebaAcceso); //Funciona correctamente
    }
}
```

Private

Permite acceder a los miembros exclusivamente desde la clase o struct que los contiene.


```

public class EjemploPrivate
{
    private string PruebaAcceso { get; set; }

    public EjemploPrivate(){
        PruebaAcceso = "funciona";//Funciona
    }
}

class Program
{
    static void Main(string[] args)
    {
        EjemploPrivate ejemplo = new EjemploPrivate();
        Console.WriteLine(ejemplo.PruebaAcceso); //Da un error
    }
}

```

internal

Permite acceder desde el mismo proyecto o assembly pero no desde uno externo.

Por ejemplo, si tenemos una librería, podremos acceder a los elementos internal desde la propia librería, pero si referenciamos a esa librería desde otro proyecto no podremos acceder a ellos.

```

//Libreria externa (Distinto proyecto|Assembly)
public class EjemploInternalLibreria
{
    internal string PruebaAcceso { get; set; }
}

class EjemploImprimirInternal
{
    public void Imprimir()
    {
        EjemploInternalLibreria ejemplo = new EjemploInternalLibreria();
        Console.WriteLine(ejemplo.PruebaAcceso); //Funciona
    }
}

//proyecto principal
class Program
{
    void Imprimir()
    {
        EjemploInternalLibreria ejemplo = new EjemploInternalLibreria();
        Console.WriteLine(ejemplo.PruebaAcceso); // Error. Al estar en otro proyec
    }
}

```

```
}  
}
```

protected

Podremos acceder a los elementos desde la misma clase, o desde una que deriva de ella.

```
class EjemploProtected  
{  
    protected string PruebaAcceso { get; set; }  
}  
  
class claseHerencia : EjemploProtected //Herencia, osea clase hija.  
{  
    void Imprimir()  
    {  
        Console.WriteLine(PruebaAcceso); //Accedemos sin problemas ya que es una p  
    }  
}  
  
class Program  
{  
    void Print()  
    {  
        EjemploProtected ejemplo = new EjemploProtected();  
        Console.WriteLine(ejemplo.PruebaAcceso); // Error. no podemos acceder ya q  
    }  
}
```

protected internal

Combina tanto protected como internal permitiendo acceder desde el mismo proyecto o assembly o de los tipos que lo derivan.

private protected

Finalmente combinamos private y protected lo que nos permitirá acceder desde la clase actual o desde las que derivan de ella. Lo que permite referenciar métodos y propiedades en clases de las cuales heredamos.

Manejo de excepciones

En el tutorial de hoy veremos, qué es una excepción y como trabajar con ellas.

Índice

- 1 - Qué es una excepción?
- 2 - Bloque try-catch
- 3 - Especificar una excepción
- 4 - Conclusión

1 - Qué es una excepción?

Una excepción es un problema o error que aparece de una manera repentina en nuestro código mientras se está ejecutando.

Debemos controlar las excepciones, ya que si no lo hacemos causara que el programa deje de trabajar, lo que implica que su funcionamiento se detiene.

La mejor forma de evitar excepciones es controlando que estas no puedan pasar. Por ejemplo, una excepción puede ocurrir cuando intentamos leer un fichero y este fichero no existe. Por lo que para evitar la excepción tendríamos que comprobar que el fichero existe, y si existe, leer, en caso contrario no leer el fichero.

Pero desafortunadamente no podemos poner un filtro o una comprobación en cada aspecto del programa. Así que lo que tenemos que hacer es un bloque `try-catch`.

2 - Bloque try-catch

C# nos permite controlar las excepciones utilizando un bloque `try-catch`. Su funcionamiento es muy básico, ponemos código dentro del bloque `try` y si salta una excepción se ejecutará el `catch`, el cual contiene otro bloque de código. Así evitamos que el programa deje de funcionar.

Por ejemplo, no podemos dividir un número por 0 lo que hace que salte una excepción.

```
try
{
    int numero = Convert.ToInt32(Console.ReadLine());
    decimal division = 25 / numero;
}
catch (Exception ex)
{
}
```

```
Console.WriteLine(ex.Message.ToString());
```

Cor
info
ocu

la
ra



Otra opción muy común dentro de un catch es utilizar la clausula throw la cual enviara la excepción al bloque catch padre.

```
try
{
    decimal division = 25 / numero;
}
catch (Exception ex)
{
    throw;
}
```

Lo que quiere decir que si el método que hace saltar la excepción esta controlado en niveles superiores ejecutara el catch de los niveles superiores. En caso de no estarlo, pararíamos la ejecución del programa.

3 - Especificar una excepción

Como hemos visto esa excepción es muy concreta, ya que no se puede dividir un numero por cero.

Dentro de un `try-catch` podemos indicar tantos `catch` como queramos, siempre y cuando el tipo de dato (`Exception`) sea diferente, como vimos con los constructores, aquí también se utiliza sobrecarga de operadores.

Para nuestro ejemplo, otra posibilidad es que el numero que introducimos sea una palabra y no un número, si queremos comprobar esa excepción en concreto, debemos especificarla en el catch como vemos a continuación.

```
Console.WriteLine("Introduce un numero");

try
{
    int numero = Convert.ToInt32(Console.ReadLine());
    decimal division = 25 / numero;
} catch (FormatException ex)
{
    Console.WriteLine("El valor introducido no era un numero entero");
}
catch(DivideByZeroException ex)
{
    Console.WriteLine("No es posible dividir por 0");
}
catch (Exception ex)
{
    throw;
}
```

Como podemos observar si el valor introducido no es un número, la ejecución del programa continuara en el `catch(FormatException)`, si ese valor es 0 continuara en `catch(DivideByZeroException)` y si la excepción es cualquier otra saltara al `catch(Exception)` el cual es el bloque por defecto.

4 - Conclusión

El manejo de excepciones es algo primordial en el día a día laboral, por eso hay que tenerlo muy en cuenta cuando realizamos nuestros proyectos personales, o mientras estudiamos, ya que le dará una robustez al sistema ante fallos y problemas catastróficos.

Parámetros por valor y referencia

Índice

- 1 - parámetros por valor
- 2 - Parámetros por referencia
 - 2.1 - Palabra reservada ref
 - 2.2 - Palabra reservada out
 - 2.3 - Palabra reservada In
- 3 - El caso especial de objetos
- 4 - Datos finales
- 5 - Cuando utilizar ref u out?
- 6 - Código

1 - Parámetros por valor

Cuando tenemos un método le pasamos un parámetro, en verdad no estamos mandando ese parámetro, estamos mandando una copia de este, lo que significa, que fuera del método, el valor de la variable seguirá siendo el mismo. Incluso si modificamos su valor internamente.

```
static void Main(string[] args)
{
    int valorActual = 10;

    Actualizar(valorActual);
    Console.WriteLine($"el valor es: {valorActual}");//imprime 10

    Console.ReadKey();
}

//Comportamiento normal.
public static void Actualizar(int valor)
{
    valor += 5;
    Console.WriteLine($"el valor es: {valor}"); //imprime 15
}
```

`valorActual` sigue valiendo 10 después de finalizar el método `Actualizar()` ya que el valor únicamente se modifica para ese método.

2 - Parámetros por referencia

Puede llegar el momento de que necesitemos actualizar el valor de los datos. Para ello pasaremos los parámetros por referencia, y lo haremos utilizando las palabras

reservadas `ref` y `out`. Cuando utilizamos `ref` y `out` no almacenamos en la variable el valor de esta, sino que almacenamos la posición de memoria a la que apunta por lo que, si modificamos el valor, también cambiamos el valor de la original.

2.1 - Palabra reservada ref

Utilizar `ref` significa que vas a pasar el valor por referencia a un método, lo que implica que el valor va a cambiar dentro del método. Además, si deseamos utilizar `ref` la variable tiene que estar inicializada con anterioridad.

Finalmente debemos indicar tanto en el método como en la llamada al método que vamos a pasar un valor por referencia.

```
static void Main(string[] args)
{
    int valorActual = 10;

    ActualizarRef(ref valorActual); //pasar por referencia
    Console.WriteLine($"el valor es: {valorActual}"); // imprime 12

    Console.ReadKey();
}

//Actualizar por referencia
public static void ActualizarRef(ref int valor)
{
    valor += 2;
}
```

2.2 - Palabra reservada out

Utilizar `out` significa que la asignación del valor de esa variable está dentro del método al que se llama. No es necesario inicializar el valor de la variable, aunque si debemos instanciarla.

```
static void Main(string[] args)
{
    int valorActual;
    ActualizarOut(out valorActual);
    Console.WriteLine($"el valor es: {valorActual}"); // imprime 13

    Console.ReadKey();
}

//Crear utilizando out
```

```
public static void ActualizarOut(out int valor)
{
    valor = 13;
}
```

2.3

La

or

lo c

```
public static void ActualizarIn(in int valor)
{
    valor += 5; //Da Error
    Console.WriteLine($"el valor es: {valor}");
}
```

3 - El caso especial de objetos

Cuando pasamos un Objeto/Tipo creado por nosotros mismos, este siempre se pasa por referencia. Con lo que, si lo actualizamos dentro de la función, se actualizara fuera de ella.

```
class Program
{
    static void Main(string[] args)
    {
        ObjEjemplo ejemploValor = new ObjEjemplo(10);
        ActualizarObj(ejemploValor);
        Console.WriteLine($"el valor es: {ejemploValor.Entero}"); // imprime 25

        Console.ReadKey();
    }

    public static void ActualizarObj(ObjEjemplo obj)
    {
        obj.Entero = 25;
    }
}

public class ObjEjemplo
{
    public int Entero { get; set; }

    public ObjEjemplo(int entero)
```



```
[ {
    Entero = entero;
}
```

- 4 - I
- De
 - va
 - No

5 - (

Util

mé

Des

de

6 - (

El c

PRELIMINAR

Recursividad en programación

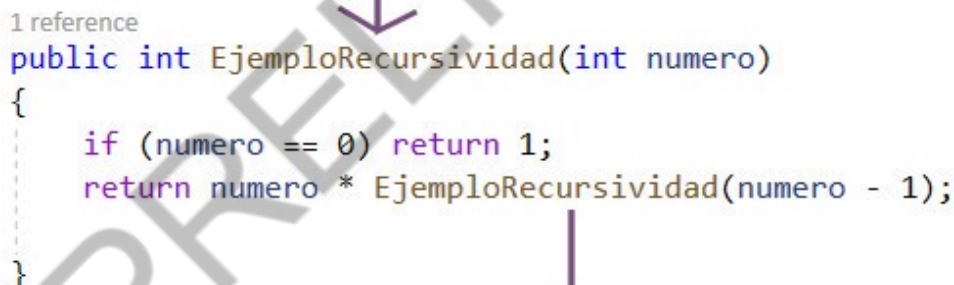
Índice

- 1 - Qué es la recursividad en programación?
- 2 - Cuando utilizar recursividad?
- 3 - La pila de recursión

En el post de hoy trataremos un tema muy importante dentro de la programación, como es, la recursividad, es cierto que es un termino que cuando lo estudias por primera vez cuesta, pero es muy sencillito. Hay que decir que no está destinado únicamente al entorno .Net, sino que sirve para todos los lenguajes de programación.

1 - Qué es la recursividad en programación?

La recursividad es un concepto que se indica cuando un método se llama a si mismo. Cuando creamos un método recursivo debemos tener en cuenta que este tiene que terminar por lo que dentro del método debemos asegurarnos de que no se está llamando a si mismo todo el rato, Lo que quiere decir que el ciclo es finito.



```
1 reference
public int EjemploRecursividad(int numero)
{
    if (numero == 0) return 1;
    return numero * EjemploRecursividad(numero - 1);
}
```

Debemos tener mucho cuidado cuando realizamos llamadas recursivas ya que si la utilizamos sin control podríamos desbordar la memoria del ordenador, causando que el programa se rompa.

2 - Cuando utilizar recursividad?

Como se puede apreciar de la descripción anterior. Podemos utilizar recursividad para reemplazar cualquier tipo de bucle. A pesar de ello en el mundo laboral no se utiliza demasiado, debido a que un error puede ser trágico en la memoria, así como tener una lista con millones de datos, puede hacer que utiliza mucha memoria. Aun así, la gran mayoría de las veces, utilizamos recursividad para algoritmos de búsqueda u ordenación.

Se denomina caso base a la condición de terminación de la recursividad.

Ejemplo 1

Ejemplo es calcular el factorial de un numero *multiplicar todos los números entre dos enteros. En matemáticas se expresa con n!, donde n es el último número a comprobar. Debemos empezar desde el 1.

Ejemplo utilizando un método iterativo

```
public static int FactorialIterativo(int numero)
{
    int i, resultado = 1;
    for(i=1; i <= numero; i++)
    {
        resultado = resultado * i;
    }
    return resultado;
}
```

El ejemplo utilizando método recursivo

```
public static int FactorialRecursivo(int numero)
{
    if (numero == 0) return 1;
    return numero * FactorialRecursivo(numero - 1);
}
```

3 - La pila de recursión

La memoria de un ordenador se divide en 4 segmentos

- Segmento de código: almacena las instrucciones del programa en código máquina.
- Segmento de datos: almacena las variables estáticas o constantes.
- Montículo: almacena las variables dinámicas
- Pila del programa: Parte destinada a las variables locales y parámetros de la función que se está ejecutando.

Cuando llamamos a una función (o método) el proceso es el siguiente

Se reserva espacio en la pila para los parámetros de la función y sus variables locales.

Se guarda en la pila la dirección de la línea del código desde donde se ha llamado al método.

Se almacenan los parámetros de la función y sus valores en la pila.

Finalmente se libera la memoria asignada en la pila cuando la función termina y se vuelve a la llamada de código original.

En cambio, cuando la función es recursiva:

- Cada llamada genera una nueva llamada a una función con los correspondientes objetos locales.
- Volviéndose a ejecutar completamente, hasta la llamada a si misma. Donde vuelve a crear en la pila los nuevos parámetros y variables locales. Tantos como llamadas recursivas generemos.
- Al terminar, se van liberando la memoria en la pila, empezando desde la última función creada hasta la primera, la cual será la última en liberarse.

Live coding serpiente

El Post de hoy consiste en un juego muy común que todos conocemos de los antiguos Nokia.

La idea de este ejercicio es ver o trabajar la gran mayoría de elementos y técnicas de programación que hemos estado viendo a lo largo del .

Índice

- 1 - Análisis
- 2 - Código
 - 2.1 - Entidad tablero
 - 2.2 - Entidad Serpiente
 - 2.3 - Entidad Caramelo
- 3 - Lógica del programa
 - 3.1 - Serpiente se mueve
 - 3.2 - Serpiente come
 - 3.3 - Moverse en diferentes direcciones
 - 3.4 - Dibujar el caramelo
 - 3.5 - Morir
 - 3.6 - Juntar las partes

El Juego de la serpiente o snake

En este post veremos una solución a como podemos realizar este juego en .net

1 - Análisis

Lo primero de todo realizaremos un análisis de los requisitos del juego que son a groso modo:

- Tenemos un tablero o un terreno donde podemos mover la serpiente
- La serpiente se puede mover
 - Arriba
 - Abajo
 - Derecha
 - Izquierda
- Caramelos que salen en la pantalla para comerselos y sumar puntos

Por lo tanto si convertimos las entidades reales a entidades de código, **como vimos en el video de objetos y clases**, obtendremos lo siguiente:

- . Clase Tablero
 - Propiedades:
 - i. Altura
 - ii. Anchura
 - Métodos:
 - i. Dibujar el tablero
- . Clase serpiente
 - Propiedades:
 - i. Cola = Lista de posiciones
 - ii. Dirección actual
 - iii. Puntos
 - Métodos
 - i. Morir
 - ii. Moverse
 - iii. Comerse el caramelo
- . Clase Caramelo

- Aparece

Además de estas clases, quizá debamos crear alguna adicional, como por ejemplo una clase `Util` que nos dibuje por pantalla una línea o caracter, ya que es una acción que vamos a repetir constantemente.

```
static class Util
{
    public static void DibujarPosicion(int x, int y, string caracter)
    {
        Console.SetCursorPosition(x, y);
        Console.WriteLine(caracter);
    }
}
```

Como podemos observar el método `Console.SetCursorPosition(x,y)` nos ubica el cursor dentro de la terminal en el punto que deseamos.

Y como vimos en el video de **entrada salida por teclado y pantalla**, `Console.WriteLine(string)` nos escribe un texto, en la gran mayoría de casos en este ejercicio, es tan solo un carácter. En el caso de los muros es el caracter `#` o la `x` para la serpiente

2 - Código

Cuando programamos debemos pensar como si tuviéramos ese objeto, u objetos, delante, por lo que el primer elemento que pondríamos sería el tablero, así que la entidad Tablero será la primera que vayamos a crear.

2.1 - Entidad tablero

Creemos la clase tablero con las propiedades `Altura` y `Anchura`, además, ya que la utilizaremos más adelante, creamos la propiedad `ContieneCaramelo` la cual la inicializamos a `false`, ya que cuando dibujamos el tablero, de primera instancia, el caramelo no existe.

También debemos crear el método `DibujarTablero` el cual recorre la altura y la anchura para dibujar el tablero indicado.

```
public class Tablero
{
    public readonly int Altura;
    public readonly int Anchura;
    public bool ContieneCaramelo;
    public Tablero(int altura, int anchura)
    {
        Altura = altura;
    }
}
```

```

        Anchura = anchura;
        ContieneCaramelo = false;
    }

    public void DibujarTablero()
    {
        for (int i = 0; i <= Altura; i++)
        {
            Util.DibujarPosicion(Anchura, i, "#");
            Util.DibujarPosicion(0, i, "#");
        }

        for (int i = 0; i <= Anchura; i++)
        {
            Util.DibujarPosicion(i, 0, "#");
            Util.DibujarPosicion(i, Altura, "#");
        }
    }
}

```

2.2
Por
rec
que
y la

le
lo
X

```

public class Posicion
{
    public int X;
    public int Y;

    public Posicion(int x, int y)
    {
        X = x;
        Y = y;
    }
}

```

Además de `List<Posicion>` deberá contener la direccion en la que se va a mover. Estas al ser solamente 4 y ser siempre las mismas, las crearemos en una enumeración

```

public enum Direccion
{
    Arriba,
    Abajo,
    Izquierda,

```

```
public class Serpiente
{
    List<Posicion> Cola { get; set; }
    public Direccion Direccion { get; set; }
    public int Puntos { get; set; }

    public Serpiente(int x, int y)
    {
        Posicion posicionInicial = new Posicion(x, y);
        Cola = new List<Posicion>() { posicionInicial };
        Direccion = Direccion.Abajo;
        Puntos = 0;
    }

    public void DibujarSerpiente()
    {
        foreach (Posicion posicion in Cola)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Util.DibujarPosicion(posicion.X, posicion.Y, "x");
            Console.ResetColor();
        }
    }

    public bool ComprobarMorir(Tablero tablero)
    {
        throw new NotImplementedException();
    }

    public void Moverse(bool haComido)
    {
        throw new NotImplementedException();
    }

    public bool ComeCaramelo(Caramelo caramelo, Tablero tablero)
    {
        throw new NotImplementedException();
    }
}
```

```
[
}
```

Fin.
des
cua
Cor
e
cóc
que
El r

y
la
la
El
as

2.3
No
con
ent

os
la

```
public class Caramelo
{
    public Posicion Posicion { get; set; }

    public Caramelo(int x, int y)
    {
        Posicion = new Posicion(x, y);
    }

    public void DibujarCaramelo()
    {
        Console.ForegroundColor = ConsoleColor.Blue;
        Util.DibujarPosicion(Posicion.X, Posicion.Y, "O");
        Console.ResetColor();
    }

    public static Caramelo CrearCaramelo(Serpiente serpiente, Tablero tablero)
    {
        throw new NotImplementedException();
    }
}
```


Ahora que ya tenemos las entidades definidas, incluso cuando algún método esta sin implementar, debemos decidir como será la lógica para que la serpiente se pueda mover y vaya comiendo caramelos.

Para ello vamos a la clase program, que como recordamos **del primer post**, es la clase que se ejecuta al ejecutar una aplicación de consola.

Para ello debemos imprimir tanto el `tablero` como la `serpiente` en primer lugar. Pero si no incluimos nada más, la serpiente se quedará estática. Como recordamos tenemos la propiedad `Direccion`. Que nos indica en qué dirección se va a mover la serpiente. Y para que se mueva debemos introducirla en un **bucle** `while` o `do while`.

```
static void Main(string[] args)
{
    Tablero tablero = new Tablero(20, 20);

    Serpiente serpiente = new Serpiente(10, 10);
    Caramelo caramelo = new Caramelo(0, 0);
    bool haComido = false;

    do
    {
        Console.Clear();
        tablero.DibujarTablero();
        serpiente.Moverse(haComido);
        haComido = serpiente.ComeCaramelo(caramelo, tablero);
        serpiente.DibujarSerpiente();

    } while (serpiente.ComprobarMorir(tablero));

    Console.ReadKey();
}
```

Debemos inicializar tanto el `tablero` como la `serpiente` como el primer `caramelo` fuera del bucle, ya que si está dentro del mismo estaríamos empezando desde 0 todo el rato.

Además, la primera línea dentro del bucle debe ser para limpiar la consola, utilizando un `Console.Clear()` ya que esta no se limpia automáticamente.

Finalmente, de esta pequeña parte de código, debemos recordar que la serpiente primero se mueve y luego come. Finalizando con dibujarla.

El primer paso del programa es moverse, para ello deberemos modificar la posición en la que nos encontramos, dependiendo de la dirección:

- **Arriba:** Y -1
- **Abajo:** Y+1
- **Derecha:** X +1
- **Izquierda:** X-1

Obtenemos la primera posición de la cola (osea la cabeza) y actualizamos su valor como podemos ver en `ObtenerNuevaPrimeraPosicion()` .

```
public void Moverse(bool haComido)
{
    List<Posicion> nuevaCola = new List<Posicion>();
    nuevaCola.Add(ObtenerNuevaPrimeraPosicion());
    nuevaCola.AddRange(Cola);

    if (!haComido)
    {
        nuevaCola.Remove(nuevaCola.Last());
    }

    Cola = nuevaCola;
}

private Posicion ObtenerNuevaPrimeraPosicion()
{
    int x = Cola.First().X;
    int y = Cola.First().Y;

    switch (Direccion)
    {
        case Direccion.Abajo:
            y += 1;
            break;
        case Direccion.Arriba:
            y -= 1;
            break;
        case Direccion.Derecha:
            x += 1;
            break;
        case Direccion.Izquierda:
            x -= 1;
            break;
    }

    return new Posicion(x, y);
}
```

Para mover el resto de la serpiente es muy sencillo. Ya que la cola entera se desplaza un valor.

Entonces, lo que hacemos es el nuevo valor que hemos obtenido, es el primer valor de una lista nueva. Y a esa lista nueva, le pegamos la cola anterior.

Por ejemplo, si tenemos 3 elementos:

Posición X	Posición Y
10	10
10	12
10	12

El siguiente movimiento es hacia arriba, ósea Y -1 el valor que obtendremos será el siguiente:

Posición X	Posición Y
10	9

Para después concatenar la lista previa, por lo que nos quedará lo siguiente:

Posición X	Posición Y
10	9
10	10
10	11
10	12

Pero esta lista tiene un valor mas del deseado. Aquí es donde entra el juego si ha comido en el bucle anterior, ya que, si ha comido esta bien, y lo dejamos como está, pero si no ha comido debemos quitar de la lista el último valor.

Con los que nos queda el siguiente resultado:

Posición X	Posición Y
10	9
10	10
10	11

Como observamos son las nuevas posiciones de la serpiente.

3.2 - Serpiente come

Para comprobar si la serpiente ha comido, debemos comprobar que el caramelo no esta en ninguna posición de la cola. No debería pasar que este se cree en una posición de la cola, pero por si acaso las comprobamos todas.

Dentro de la clase serpiente el código para comprobar si hemos comido es el siguiente:

```
public bool ComeCaramelo(Caramelo caramelo, Tablero tablero)
{
    if (PosicionEnCola(caramelo.Posicion.X, caramelo.Posicion.Y))
    {
        Puntos += 10; // sumamos puntos
        tablero.ContieneCaramelo = false; //Quitar el caramelo o generar uno nuevo
        return true;
    }
    return false;
}
```

```

public bool PosicionEnCola(int x, int y)
{
    return Cola.Any(a => a.X == x && a.Y == y);
}

```

Com
tier

10

3.3
Per
es |
Par

er

```

var sw = Stopwatch.StartNew();
while (sw.ElapsedMilliseconds <= 250)
{
    serpiente.Direccion = Util.LeerMovimiento(serpiente.Direccion);
}

```

El método `LeerMovimiento` nos devolvera la `Direccion` que hemos pulsado, si no pulsamos una nueva dirección durante 200milisegundos, devolvera automaticamente la dirección actual.

```

static Direccion LeerMovimiento(Direccion movimientoActual)
{
    if (Console.KeyAvailable)
    {
        var key = Console.ReadKey().Key;

        if (key == ConsoleKey.UpArrow && movimientoActual != Direccion.Abajo)
        {
            return Direccion.Arriba;
        }
        else if (key == ConsoleKey.DownArrow && movimientoActual != Direccion.Arriba)
        {
            return Direccion.Abajo;
        }
        else if (key == ConsoleKey.LeftArrow && movimientoActual != Direccion.Derecha)
        {
            return Direccion.Izquierda;
        }
        else if (key == ConsoleKey.RightArrow && movimientoActual != Direccion.Izquierda)
        {
            return Direccion.Derecha;
        }
    }
}

```

```
return movimientoActual;
```

```
}
```

Debemos evitar ir en dirección contraria a la que vamos actualmente, o eso causara que choquemos y muramos contra nosotros mismos.

3.4 - Dibujar el caramelo

Dibujar el caramelo es muy sencillo, realizaremos una acción similar a cuando dibujamos la serpiente o los muros. La única diferencia es que, para dibujar el caramelo, debemos estar seguros de dos factores.

- Es totalmente aleatorio
- No se encuentra en los muros ni en la serpiente.

```
public void DibujarCaramelo()
{
    Console.ForegroundColor = ConsoleColor.Blue;
    Util.DibujarPosicion(Posicion.X, Posicion.Y, "O");
    Console.ResetColor();
}

public static Caramelo CrearCaramelo(Serpiente serpiente, Tablero tablero)
{
    bool carameloValido;
    int x,y;

    do
    {
        Random random = new Random();
        x = random.Next(1, tablero.Anchura - 1);
        y = random.Next(1, tablero.Altura - 1);
        carameloValido = serpiente.PosicionEnCola(x, y);

    } while (!carameloValido);

    tablero.ContieneCaramelo = true;
    return new Caramelo(x, y);
}
```

3.5 - Morir

Para el apartado morir, comprobaremos únicamente dos aspectos.

- La cabeza de la serpiente (o primera posición de la lista) está en una pared.
- La cabeza de la serpiente está en el resto de la cola de la serpiente.

```
public void ComprobarMorir(Tablero tablero)
{
```

```

//Si nos chocamos contra nosotros
Posicion primeraPosicion = Cola.First();

EstaViva = !((Cola.Count(a => a.X == primeraPosicion.X && a.Y == primeraPosicion
    || CabezaEstaEnPared(tablero, Cola.First())));
}

//si la primera posicion esta en cualquiera de los muros, morimos.
private bool CabezaEstaEnPared(Tablero tablero, Posicion primeraPoisicon)
{
    return primeraPoisicon.Y == 0 || primeraPoisicon.Y == tablero.Altura
        || primeraPoisicon.X == 0 || primeraPoisicon.X == tablero.Anchura;
}

```

3.6 - Juntar las partes

Finalmente, para que todo funcione correctamente debemos juntar cada una de las partes, en un orden con sentido. Para ello dentro de la clase `main` dividiremos el código de la siguiente manera.

Primero, como hemos visto antes inicializamos las variables que necesitamos, ellas incluye, el `tablero`, la `serpiente`, el `caramelo` y la variable `haComido`.

```

Tablero tablero = new Tablero(20, 20);

Serpiente serpiente = new Serpiente(10, 10);
Caramelo caramelo = new Caramelo(0, 0);
bool haComido = false;

```

Una vez tenemos todo inicializado, debemos pasar a dibujarlo, recordamos que esta todo en un bucle `do While`. Dentro del bucle, debemos primero de todo, limpiar la pantalla, y después dibujarlo todo. siempre y cuando la serpiente este viva.

Podemos hacer la comprobación de si esta viva tanto al principio del búcle, como al final del mismo. no importa. pero si lo hacemos al principio, debemos ejecutar los movimientos, creacion de la serpiente y el caramelo únicamente si seguimos vivos. En caso opuesto, mostraremos el mensaje de que hemos muerto y una puntuación.

Siendo el siguiente código el resultado de la clase `main`

```

static void Main(string[] args)
{
    Tablero tablero = new Tablero(20, 20);

    Serpiente serpiente = new Serpiente(10, 10);

```

```

Caramelo caramelo = new Caramelo(0, 0);
bool haComido = false;

do
{
    Console.Clear();
    tablero.DibujarTablero();
    //movemos y comprobamos si ha comido en el turno anterior.
    serpiente.Moverse(haComido);

    //Comprobamos si se ha comido el caramelo
    haComido = serpiente.ComeCaramelo(caramelo, tablero);

    //Dibujamos serpiente
    serpiente.DibujarSerpiente();

    //Si no contiene el caramelo, instanciamos uno nuevo.
    if (!tablero.ContieneCaramelo)
    {
        caramelo = Caramelo.CrearCaramelo(serpiente, tablero);
    }

    //Dibujamos caramelo
    caramelo.DibujarCaramelo();

    //Leemos informacion por teclado de la direccion.
    var sw = Stopwatch.StartNew();
    while (sw.ElapsedMilliseconds <= 250)
    {
        serpiente.Direccion = Util.LeerMovimiento(serpiente.Direccion);
    }

} while (serpiente.ComprobarMorir(tablero));

Util.MostrarPuntuación(tablero, serpiente);

Console.ReadKey();
}

```

C:\Program Files\dotnet\dotnet.exe

```
#####
#                                           #
#                                           #
#                                           #
#                                           #
#           XXXXX                           #
#         X                                 #
#   XXXXX                                   #
#  X                                           #
#  X                                           #
#  0                                           #
#####
```


La mejor forma de representar números en c#

Hoy vamos a ver un post corto, pero creo que es importante, ya que mucha gente desconoce esta característica del lenguaje c#, y no solo eso, sino que veremos una librería que nos hace la vida más fácil para representar números grandes.

Índice

- 1 - Tipos enteros
 - 1.1 - Literales
- 2 - Asignar valores en nuestro código
 - 2.1 - Librería Numerize

1 - Tipos enteros

Todo lo que vamos a ver aquí representa los tipos numéricos enteros, osea positivos y negativos sin decimales, en términos del lenguaje C# Tenemos `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `nint` , `nuint` .

La diferencia entre `long` y `ulong` es que `long` tiene signo (positivo o negativo) y `ulong` no, lo que quiere decir que el número que puedes representar es “mas grande”, aunque siempre sera “positivo”) ya que en la versión con signo `32 bits` son positivos y 32 negativos, mientras que en la versión sin signo son `64 bits` positivos.

1.1 - Literales

Dentro de los tipos enteros también nos entran diferentes tipos, que son, los decimales, que los acabamos de ver, los valores hexadecimales, que tienen un prefijo de `0x` , o los valores binarios, que tienen un prefijo de `0b` ;

Por ejemplo si representamos el valor 2022 sería lo siguiente:

```
var hex = 0X7E6
var binary = 0b11111100110
```

2 - Asignar valores en nuestro código


El punto principal por el que creo este post es por la siguiente información. Cuando representamos un número como 2022 es fácil de leer en el código, pero qué pasa si nuestro número es más grande? por ejemplo `43209412016` ; ese número a simple vista es imposible de saber cual es.

C# nos provee de una característica que nos permite representar los números de una forma más sencilla, la cual es añadir la barra baja (`_`) entre números. entonces el siguiente número quedaría de la siguiente manera.

```
long valor = 43_209_412_016
```

2.1 - Librería Numerize

Pero para mí, esto no es suficiente, he creado una librería que nos permite indicar ese mismo número pero en inglés, lo cual, lo hace mucho más fácil de leer.

La librería en cuestión es **Numerize** y te invito a que le des una buena estrella  . Por supuesto está en **Nuget**.

Con la librería podemos escribir el número en inglés sin preocuparnos de barras bajas o de no saber a simple vista qué número es.

Por ejemplo con el número anterior, podemos hacer lo siguiente:

```
long value = Numerize.Fourty.Three().Billion()  
                .Two().Hundred().Nine().Milion()  
                .Four().Hundred().Twelve().Thousand()  
                .Sixteen();
```

Y con el uso de **operadores implícitos** también convierte el número a texto:

```
Numerize numerize = new Numerize();  
string result = numerize.Fourty().Three().Billion()  
                .Two().Hundred().Nine().Milion()  
                .Four().Hundred().Twelve().Thousand()  
                .Sixteen();
```

Y nos lo convierte en lo siguiente (en inglés): `forty three billion two hundred nine million four hundred twelve thousand sixteen` .

Lo cual es mucho más amigable

PRELIMINAR