Curso de Programación .NET nivel I

Unidad 2

“El lenguaje C#”

Contacto: consultas@elearning-total.com
Web: www.elearning-total.com

Índice

Introducción

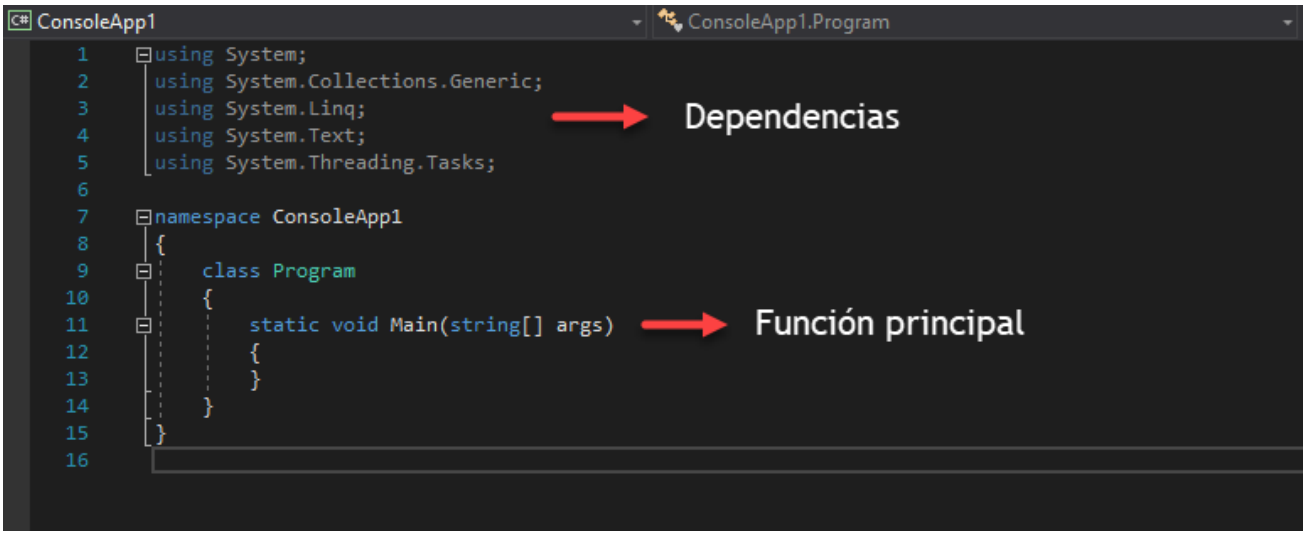
En esta unidad vamos a ver los conceptos fundamentales de C# para poder programar aplicaciones con este lenguaje. Si ya sabés algo de programación va a ayudar pero no es requisito necesario para poder completar esta unidad o el curso.

C# es uno de los lenguajes más populares del mundo y con buenas razones, ofrece un gran poder y flexibilidad como a su vez te protege de errores de programación de otros lenguajes.
C# y .NET ofrecen en sus últimas versiones muchas ayudas y atajos al programador que vienen muy bien a la hora de ponerse a codear.

Pero tranquilo vamos a ir de a poco, en este capitulo vamos a empezar explicando los conceptos y las estructuras básicas junto a ejemplos y aplicaciones en la vida real.

Cuando terminemos este capitulo vamos a estar listos para meternos en el mundo de la programación orientada a objetos por medio de C# y vamos a utilizar los conceptos básicos aprendidos en esta unidad.

Estructura de código C#



En la imagen vemos la estructura inicial cuando creamos una nueva aplicación consola lista para comenzar a programar. Quiero hacerles notar dos cosas importantes acá:

- **Dependencias:**
  Las dependencias son código ya creado de .NET o puede ser también por nosotros mismos y para decirle a Visual Studio que queremos usar tal o cual dependencia en nuestro código fuente debemos declararlo así:

```
using System.Threading.Tasks;
```

  Ya vamos a ver algunas dependencias conocidas para hacer cosas interesantes con C#. Las DLL se agregan al código fuente de esta manera también, siempre y cuando estén en nuestras referencias (Recordar Unidad 1)

- **La función Main**
  Es la función de partida en una aplicación tipo consola, desde aquí comienza y termina el código o mejor dicho las sentencias. Dentro de los corchetes **{ }** de `Main` vamos a ir poniendo nuestro código para ir ordenandole a nuestra Computadora que hacer.

```
static void Main(string[] args)
{
    Console.WriteLine("Hola Mundo");
}
```

Sentencias y Expresiones

Muchas veces leyendo foros o manuales vamos a ver que para cada linea de código a veces se habla de Sentencias y otras veces de expresiones, entonces ¿en qué se diferencian?

- **Sentencias (Statement)**
  Cada linea de código de nuestro programa que le dice que hacer a la computadora es una sentencia.

- **Expresiones**
  Son un tipo especial de sentencia, y son las sentencias que devuelven un valor, ya sea por referencia o por valor (esto lo veremos más adelante)

  Ejemplos:

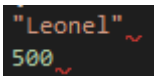
```
int i = 3 + 2;
var hola = "Hola";
```

Noten que en C# las instrucciones terminan con un ; (punto y coma) A diferencia de otros lenguajes que el salto de linea es suficiente. Esto me permite en C# si quisiera juntar todo en una misma linea.

Olvidarse de poner el ; detrás de cada instrucción es un error común cuando se programa en C#, aún a personas con experiencia.

Literales

Los literales los hemos estado usando pero no nombramos todavía. Los literales son valores que se ingresan por teclado y son de valor fijos. Además de un valor tienen un tipo asociado que veremos a continuación.



Variables

Las variables son estructuras donde se guarda un valor, el cual puede ir modificándose a lo largo del tiempo. El uso más obvio de las variables es de evitar la repetición por ejemplo:

```
var MiVariable = "Leonel";
Console.WriteLine("Hola " + MiVariable);
Console.WriteLine(MiVariable + "es tu nombre no?");
```

En este caso podría repetir mi nombre en ambas sentencias sin mayor esfuerzo, pero si nos vamos a un escenario muchos más complejos de múltiples variables y miles de lineas ahí la cosa cambia y la probabilidad de olvidar de modificar alguna linea en caso de ser necesario aumenta.

Tipos

Las variables tienen distintos tipos y dependiendo de su tipo se pueden hacer o no ciertas operaciones con el dato.

- **String:** corresponden a tipos de datos de texto a los cuales se los puede recortar, pasar a Mayúsculas, imprimir en pantalla como el "hola mundo", entre otras funciones de texto. Los tipos String se declaran literalmente entre doble corchetes `"así"`.
  Este tipo de dato es uno de los más importantes ya que es único tipo de dato que se permite mostrar en pantalla y el único que nuestro usuario realmente ve. Los demás tipos de datos deben convertirse a String antes de ser mostrado en pantalla.
- **int**: corresponden al tipo numérico entero y se pueden aplicar toda operación aritmética para enteros (suma, resta, potencia) sobre ellos. Para operaciones con decimales hay tipos especiales para tal motivo como el **float** o el **double**.
  Además dependiendo hasta que número queremos guardar tenemos **int** de distinto tamaño (byte, short, long)

Todos los tipos de datos:

Nombre de la clase	Tipo de dato en C#	Descripción
Byte	Byte	Entero sin signo de 8 bit.
Sbyte	sbyte	Entero sin signo de 8bit (Tipo no acorde con el CLS)
Int16	short	Entero con signo de 16 bit.
Int32	int	Entero con signo de 32 bit.
Int64	long	Entero con signo de 64 bit.
Uint16	ushort	Entero sin signo de 16 bit. (Tipo no acorde con el CLS)
Uint32	uint	Entero sin signo de 32 bit. (Tipo no acorde con el CLS)
Uint64	ulong	Entero sin signo de 64 bit. (Tipo no acorde con el CLS)
Single	float	Numero con coma flotante de precisión simple, de 32 bit.
Double	double	Numero con coma flotante de precisión doble, de 64 bit.
Boolean	bool	Valor logico
Char	char	Carácter unicode de 16 bit.
Decimal	decimal	Valor decimal de 96 bit.
IntPtr	--	Entero con signo cuyo tamaño depende de la plataforma: 32 bit en plataformas de 32 bit y 64 bit en plataformas de 64 bit. (Tipo no acorde con el CLS)
UintPtr	--	Entero sin signo cuyo tamaño depende de la plataforma: 32 bit en plataformas de 32 bit y 64 bit en plataformas de 64 bit. (Tipo no acorde con el CLS)
String	string	Cadena de caracteres.

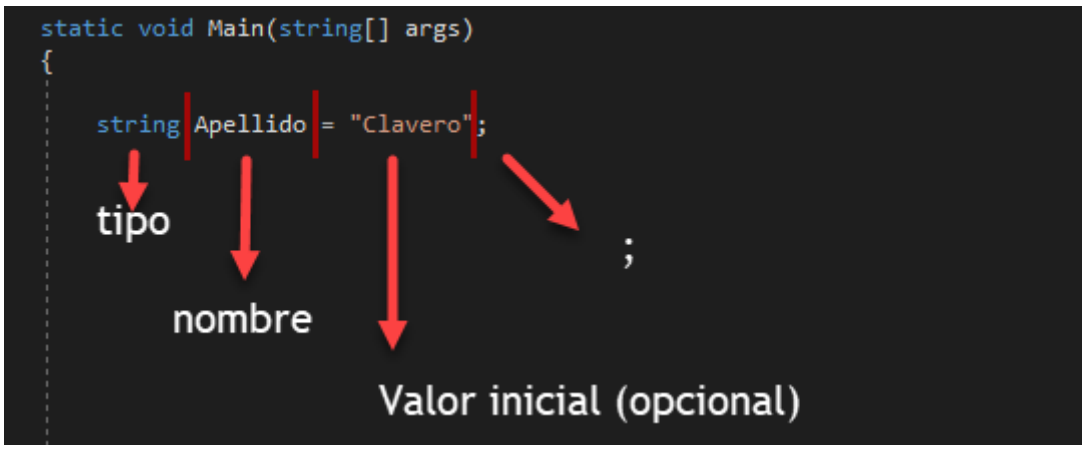
El tipo var

El tipo var es un tipo “comodín” al cual dejo que el sistema infiera el tipo.

```
var MiVariableInferida = "Leonel";
string MiVariableConTipoDefinido = "Clavero";
```

Declaración

La declaración de variables es del siguiente tipo:



si no se le asigna ningun valor inicial la variable comienza con un valor **null (núlo)** que es distinto a cadena vacía **“”** o al numero 0 en el caso de los números.

Operadores

Los operadores son símbolos que especifican que operaciones aplicarle a una expresión. Entre los más comunes tenemos los operadores de matemática, de comparación, lógicos .

```
int suma = 3 + 5;

bool masGrande = 6 > 5;
```

Acá vemos un operador suma aplicado a dos valores y en el segundo caso un operador de comparación. La segunda operación devuelve un tipo **bool** (**true** o **false)**, que es un tipo de dato bastante usado en programación como veremos en el siguiente tema.

Para ver una lista muy completa de operadores pueden referir a la documentación de .NET que es bastante extensa. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/

Condicionales

Un concepto clave en la programación es la posibilidad de dividir el flujo de tu programa dependiendo si algo es verdadero o falso (**true** o **false**) y eso es lo que hacemos con una sentencia **if** en C#.

IF

```
if (edad < 18)
{
        Console.WriteLine("Tienes que ser mayor para poder entrar");
}
```

Si la edad que ingreso es menor a 18
entonces mostrá un mensaje para que la persona sea alertada
(`Console.Read` es una función para ingresar datos string por teclado y luego uso una función para convertirlo a tipo **int,** no se mareen con esto por ahora)

ELSE

Significa "sino" y es el camino que no se siguió del **If.** En nuestro ejemplo el **else** abarcaría a todos los mayores e iguales a 18 ya que ese es el grupo que dejamos fuera en el primer paso.

```
int edad = Convert.ToInt32(Console.ReadLine());

if (edad < 18)
{
    Console.WriteLine("No podes ver esta pelicula");
}
else
{
    Console.WriteLine("Podes ver esta pelicula");
}
```

else es siempre opcional y puede ser seguido de otro **if**, otra forma de escribir lo anterior:

```
if (edad < 18)
{
    Console.WriteLine("No podes ver esta pelicula");
}
else if (edad >= 18)
{
    Console.WriteLine("Podes ver esta pelicula");
}
```

Múltiples caminos

El **else if** tiene sentido cuando tenemos que decidir entre más de 3 opciones o si 2 caminos no son lo contrario del otro.

```
if (edad < 16)
{
    Console.WriteLine("No podes ver esta pelicula");
}
else if ((edad >= 16) && (edad <= 18))
{
    Console.WriteLine("Podes ver esta pelicula en compañia de un mayor");
}
else
{
    Console.WriteLine("Ya sos grandulón, podes pasar");
}
```

Nota especial en la linea 4 que utilizamos un nuevo operador, el operador lógico and **(&&)** que se utiliza para operaciones con **bool.** Se entiende mejor si lo pasamos al español: Si la edad es mayor igual a 16 **Y** es menor igual a 18 entonces. También presten atención a los paréntesis adicionales.

Múltiples caminos con switch

Hay otra sentencia para múltiples decisiones llamada **switch** que es una opción para evitar utilizar muchos **if else.**

Este tipo de sentencia es principalmente aprovechable cuando tenemos muchas opciones singulares a partir de una expresión, que en nuestro caso es la que está entre paréntesis. El ejemplo anterior que tiene rangos no es un buen candidato para escribirse como **switch** ya que necesitarían al menos 18 **case**.

```
Console.WriteLine("Que timbre quiere tocar 1, 2, 3? los demás van al portero:");
int timbre = Convert.ToInt32(Console.ReadLine());

switch (timbre)
{
    case 1:
        {
            Console.WriteLine("Tocaste el timbre de Viviana");
            break;
        }
    case 2:
        Console.WriteLine("Tocaste el timbre de Roberto");
        break;
    case 3:
        {
            Console.WriteLine("Tocaste el timbre de Viviana");
            break;
        }
    default:
        Console.WriteLine("Tocaste el timbre del portero");
        break;
}
```

Como ven los corchetes son opcionales y luego de cada **case** deben escribir un **break;**

Una vez terminada la ejecución del código dentro del case el flujo del programa se va del **switch** y sigue su ejecución.

Los códigos de condicionales están disponibles para bajarlos desde el campus, son los proyectos **U2Condicionales1** y **U2Condicioanes2** dentro de la solución de la unidad.

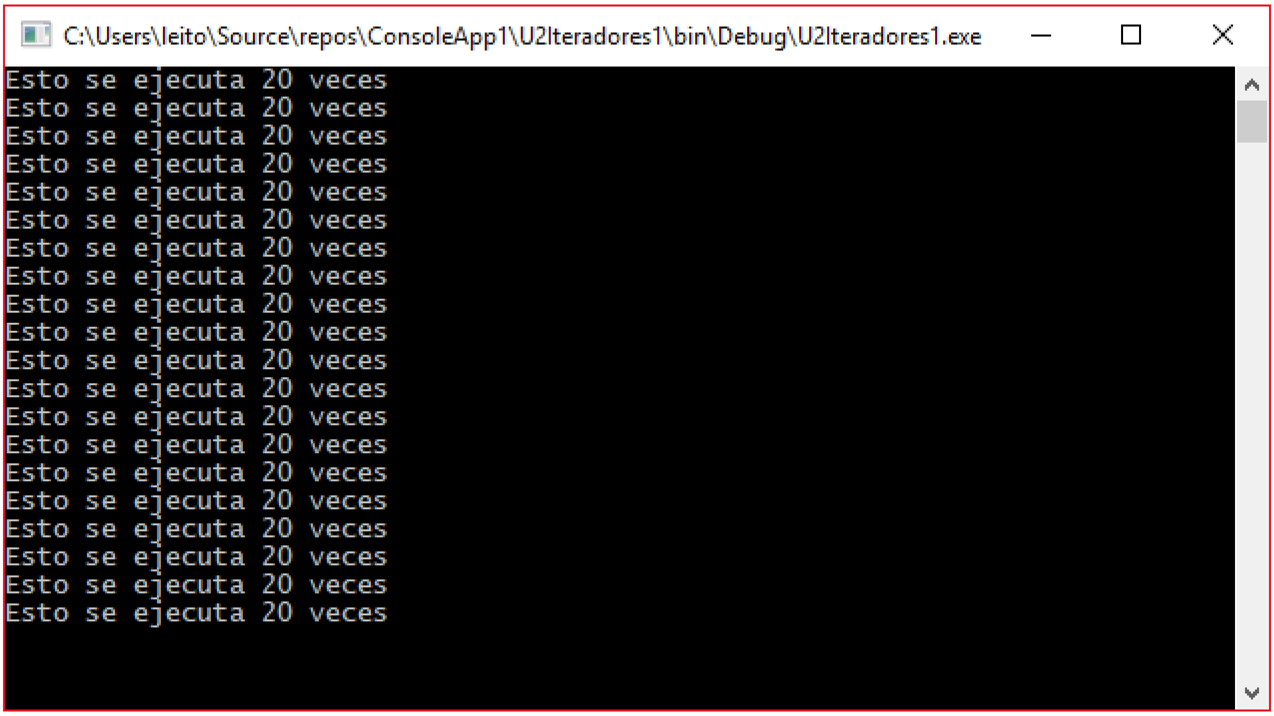
Ciclos

El ciclo For

El **For** permite ejecutar una instrucción o un conjunto de instrucciones varias veces.
Una ejecución repetitiva de sentencias se caracteriza por:

1. La prueba de condición antes de cada repetición
2. La o las sentencias que se repiten.

```
1 → for (int i = 0; i < 20; i++)
    {
2 →     Console.WriteLine("Esto se ejecuta 20 veces");
    }
```



En la imagen vemos un código que repite lo mismo unas 20 veces.

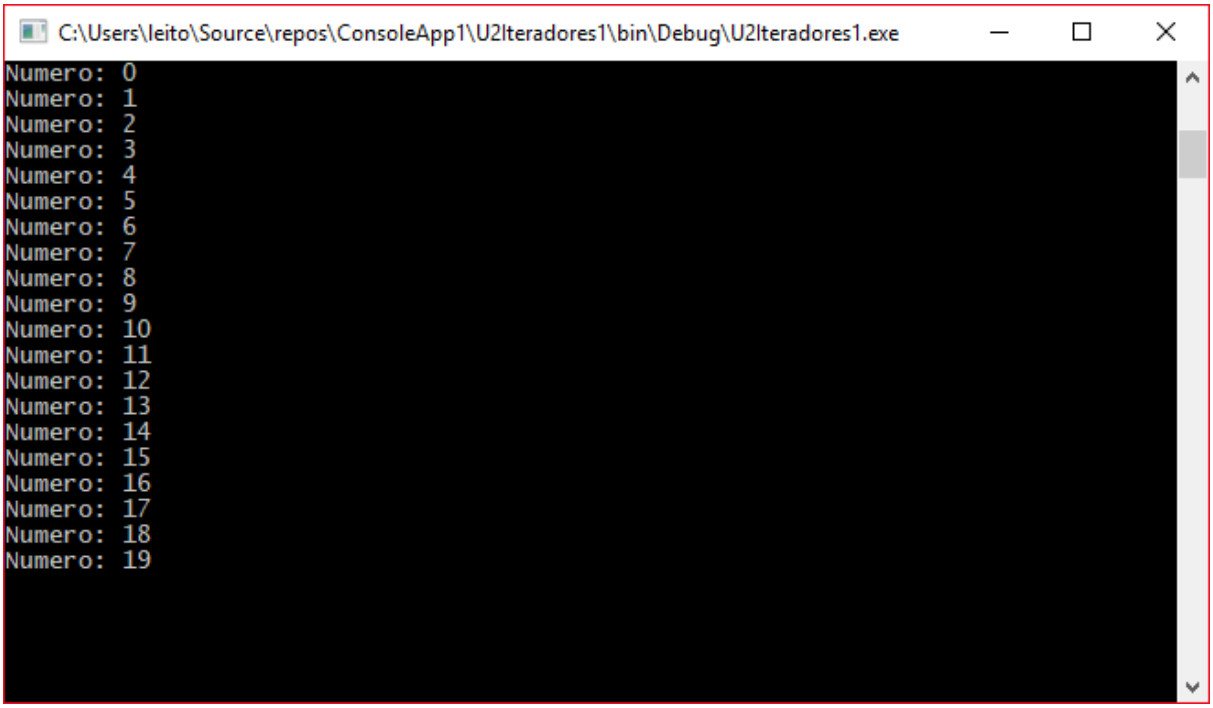
La primera parte de la estructura **for** consta de 3 sentencias:

```
  //      1          2      3
for (int i = 0; i < 20; i++)
{
    Console.WriteLine("Esto se ejecuta 20 veces");
}
```

1. Caso base:
   El código este se ejecuta la primera vez, en la imagen la variable **i** es la variable contador que empieza arbitrariamente desde 0 **(i = 0)**
2. Condición de corte:
   La condición para que se termine el ciclo de repetición, en el ejemplo quiero que llegue hasta 19 **(i < 20)**
3. Sentencia de repetición:
   la sentencia que se repite en cada ciclo relacionada a la prueba de condición. En este caso estamos usando **i++** que es un operador especial y no es más que una abreviación de **i = i + 1,** lo cual pasado en castellano: al valor que tiene hoy i sumale 1 y asignalo a i.

Obviamente es posible y deseable en la mayoría de los casos usar la variable de repetición:

```
for (int i = 0; i < 20; i++)
{
    Console.WriteLine("Numero: "+ i.ToString());
}
```



En la práctica el principal uso del **for** es el de recorrer estructuras de datos en los cuales tengo un índice y quiero realizar acciones a cada miembro de esa estructura.

Por ejemplo: Tengo una lista de productos y les tengo que actualizar su precio, en vez de programar un código de actualización para cada uno, recorro la lista de productos con un **for** y le aplico los cambios.

El ciclo While

La estructura repetitiva **for** es muy útil cuando sabemos la cantidad de elementos que queremos modificar. Cuando esa cantidad no la podemos saber de antemano tenemos otra estructura muy similar llamada **While.**

Imaginemos que queremos realizar acciones repetidamente todos los días antes de las 6 de la tarde. En ese caso el **for** no tendría sentido y el **while** sería naturalmente su reemplazante.

Vamos a reescribir el ejemplo del **for** que es un caso particular del **while**.

```
int i = 0;
while (i < 20)
{
    Console.WriteLine("Numero: " + i.ToString());
    i++;
}
```

Fíjense que los 3 elementos del **for** siguen estando simplemente cambiaron de posición.

Ahora veamos un uso del **while** donde tenga más sentido:

```
while (DateTime.Now.DayOfWeek == DayOfWeek.Monday)
{
    Console.WriteLine("Todavia es lunes, presiona enter para ver si esto cambió");
    Console.ReadLine();
}
```

Lo que está entre paréntesis por ahora no tiene mucha importancia, simplemente sepan que estoy preguntando al sistema si hoy es lunes o no.

En este caso no tengo una cantidad contable de elementos como para usar un **for**, simplemente estoy preguntando si hoy es lunes, entonces si lo es repito el código entre los corchetes.

(La sentencia ReadLine la agrego para simular el presionado del enter y evitar que esto se ejecute millones de veces seguidas colgando su computadora)

El ciclo do While

La estructura do while es otra estructura repetitiva la cual ejecuta al menos una vez su bloque repetitivo, a diferencia del while o del for que podían no ejecutar el bloque.
El ciclo do While se utiliza cuando sabemos de antemano que por lo menos una vez se ejecutará el bloque repetitivo.
La condición de la estructura está abajo del bloque a repetir, a diferencia del while o del for que está en la parte superior.

```
int numero = 100;
do
{
    Console.WriteLine("Ingrese un numero mayor a 10 para romper el ciclo: ");
    numero = int.Parse(Console.ReadLine());
}
 while (numero < 10);
```

En el ejemplo de arriba, si esto fuera un while común, nunca entraría al ciclo ya que de entrada 100 no es menor que 10; Pero como esta programado con el do while, si o si está obligado a entrar al ciclo y entonces seguir su ejecución hasta que el usuario se le ocurra poner un numero mayor a 10.

Funciones

A lo largo de este curso ya estuvimos usando funciones, sin ir mas lejos estuvimos utilizando la función `WriteLine()` para imprimir texto en pantalla.

Las funciones son una estructura disponible en muchos lenguajes de programación para agrupar código con un sentido lógico. Este agrupamiento lo que me permite es que nuestro código sea mucho mas entendible y además permite evitar que repitamos el mismo código varias veces.

Podemos distinguir dos tipos de funciones

- Procedimientos
  Los procedimientos ejecutan un código a petición pero no devuelven ningún resultado por medio de la función.

```
void EscribimeElHolaMundo()
{
    Console.Write("Hola Mundo");
}
```

- Funciones
  Las funciones también ejecutan un código a petición pero ellas devuelven un resultado. Si devuelven un valor, ese valor tiene que tener un tipo, por lo tanto la función también tiene un tipo.

```
int ElCalculoDeSiempre()
{
    return 1 + 1;
}
```

Notemos la aparición de una nueva palabra clave, el return. El return define el valor o la variable a devolver por la función.

Una vez que la función esta definida puede ser utilizada desde la función Main o mismo desde dentro de otras funciones.

Por ejemplo:

```
void EscribimeElHolaMundo()
{
    int calculo;
    calculo = ElCalculoDeSiempre();
    Console.Write("Hola Mundo y el calculo de siempre da:" + calculo.ToString());
}
```

Parámetros

Los parámetros son una manera de enviarle valores o variables a las funciones para que sean procesados por las mismas. Volviendo el ejemplo del "hola mundo", la función WriteLine recibe un parámetro de tipo texto para luego procesarlo y enviarlo a la pantalla de la consola.
Los parámetros se declaran entre los paréntesis detrás de la función junto a su tipo y nombre separados por una coma en el caso de requerir más de dos valores, veamos esto en el ejemplo de una función suma.

```
static int suma(int a, int b)
{
return a + b;
}
```

La función puede ser llamada pasandole parámetros literales o variables:

```
static void Main(string[] args)
{
    suma(1, 4);
    int a = 1;
    int b = 4;
    suma(a, b);
}
```

En C# hay dos maneras de pasar parámetros: por valor y por referencia. Pasarlos por referencia permite que los cambios que se les hagan a las variables persistan en el tiempo una vez que la función finaliza. Por el contrario pasarlos por valor no persisten en el tiempo. Para simplificar recuerden que por referencia estamos pasando un original y por valor estamos pasando una copia.
Para pasar una variable por referencia hay que agregar la palabra clave ref a la declaración de la función y también al momento de pasar la variable, por ejemplo:

```
static void SumaPorValor(int a, int b, int resultado)
{
    resultado = a + b;
}

static void SumaPorRef(int a, int b, ref int resultado)
{
    resultado = a + b;
}
```

Para llamar a estas funciones:

```
//Funcion con parametros por valor, y va a dar siempre 0
int resultado = 0;
SumaPorValor(1, 4, resultado);
Console.WriteLine("El Resultado es:", resultado);


//Funcion con parametro por referencia, va a dar la suma correcta
SumaPorRef(1, 4, ref resultado);
Console.WriteLine("El Resultado es:" ,resultado);
```

En el primer caso decimos que falla porque al pasar el parámetro *resultado* por valor (sin el **ref**) al terminar de ejecutarse la función el valor no se persiste y sigue en 0, ya que la variable *resultado* es una copia temporal que se descarta una vez finalizada la función, para que persista recuerden que tienen que enviarlo por referencia (**ref**).

Comentarios

Por último veamos un tema simple pero no menos importante ya que comentar el código es una de las actividades que todo programador debería hacer, principalmente en secciones complejas. El código debe ser comentado de manera tal que si yo mismo vuelvo a los 6 meses a ver el código, entienda perfectamente que lógica plantié en el pasado y poder seguir trabajando sin problemas.

Un comentario de código es un texto que se guarda en el archivo del código fuente pero no se compila, es decir que no agrega ni quita nada al programa final. Su principal utilidad es de brindar información acerca de cómo esta implementada nuestra aplicación a otros programadores.
Hay principalmente dos maneras de agregar comentarios en Visual Studio, la primera y más simple es por medio de la doble barra //, todo el texto que yo ingreso después de estas barras hasta el próximo renglón será tomado como comentario.

```
//esta funcion me sirve para hacer un calculo súper imposible
suma(a, ref b);
```

La segunda manera son los comentarios de múltiples líneas y esto se especifica por medio de dos operadores, el operador de apertura /* y el de clausura */; Todo el texto que este entre estos dos operadores será tomado como comentario aunque estén en distintos renglones

```
/* este es un comentario multilinea
y su único fin es enseñarles
a los estudiantes de C# como escribirlo */
```