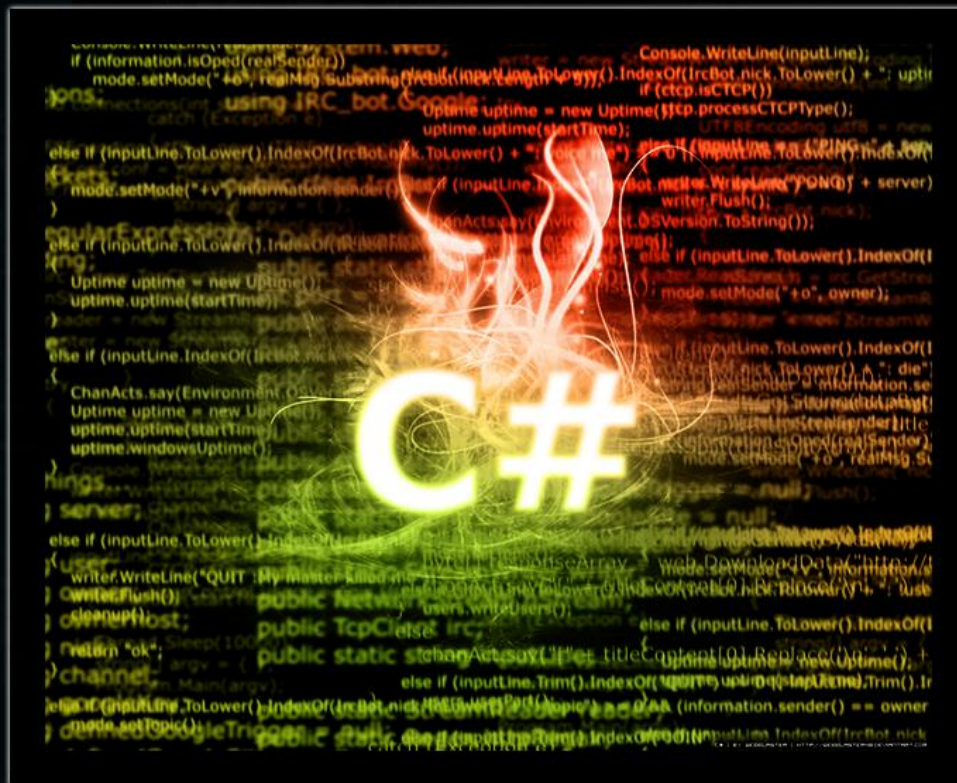


# UNDERCODE

## TALLER DE PROGRAMACION C#



## TEMAS

**INTRODUCCIÓN A C#**  
**ESQUELETO DE UN PROGRAMA**  
**VARIABLES**  
**CONSTANTES**  
**OPERADORES RELACIONALES**  
**EJERCITACIÓN**  
**Y MUCHO MÁS..!**

**TUTOR**  
**CRAZYKADE**

# **INDICE:**

- 1. Sobre este taller**
- 2. Introducción a C#**
- 3. Todo es un objeto**
- 4. Esqueleto de un programa en C#**
- 5. La clase Console**
- 6. Nuestro primer programa en C#**
- 7. Variables y Constantes**
- 8. Constantes**
- 9. Variables**
- 10. Inicialización de Variables**
- 11. El operador de asignación “=”**
- 12. Operadores Relacionales**
- 13. Ejercicio de aplicación**
- 14. Agradecimientos y Despedida**

## 1. Sobre este taller:

Hola a todos y bienvenidos a este taller de **C# .Net** en donde aprenderemos los fundamentos del lenguaje, elementos y recursos para poder desarrollar aplicaciones, finalizaremos obteniendo una base sólida y bien fundamentada respecto al **framework .Net**, su funcionamiento e implementación.

Los requisitos necesarios para poder realizar este taller son mínimos. Solo necesitaremos alguna versión de **Microsoft Visual Studio .Net**. Si lo desean, pueden

descargar la versión **express** desde: <http://www.microsoft.com/express>

Cabe aclarar que la versión **express** de **Visual Studio .Net** es muy completa la cual podrá satisfacer las necesidades que podremos tener a lo largo de este taller. Sin más palabras solo quiero agradecer a **Undercode** por brindar un espacio para aprender y hacer que el conocimiento siga siendo libre.

## 2. Introducción a C#:

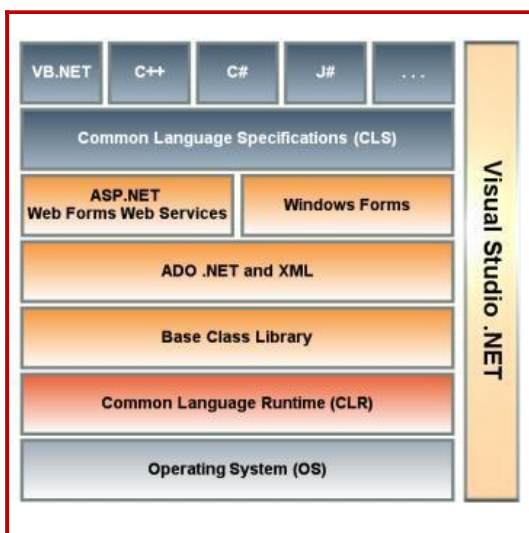
Antes de comenzar a teorizar sobre el lenguaje, veamos que dice nuestra querida Wikipedia al respecto:

*C# (pronunciado si sharp en inglés) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270). C# es uno de los lenguajes de programación diseñados para la infraestructura de lenguaje común.*

*Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.*

*El nombre C Sharp fue inspirado por la notación musical, donde '#' (sostenido, en inglés sharp) indica que la nota (C es la nota do en inglés) es un semitono más alta, sugiriendo que C# es superior a C/C++. Además, el signo '#' se compone de cuatro signos '+' pegados.*

*Aunque C# forma parte de la plataforma .NET, ésta es una API, mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Ya existe un compilador implementado que provee el marco Mono - DotGNU, el cual genera programas para distintas plataformas como Windows, Unix, Android, iOS, Windows Phone, Mac OS y GNU/Linux.*



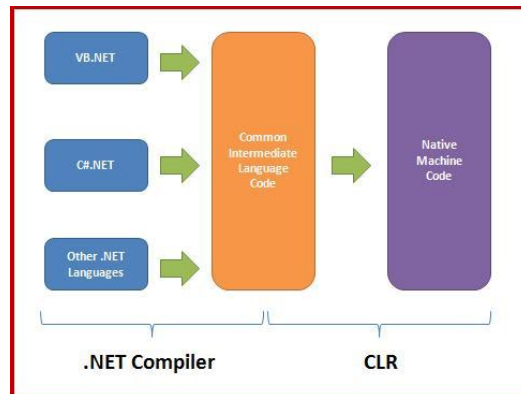
**C#** es un lenguaje sumamente potente y flexible, ya que cuenta con todas las virtudes de **C++** sumadas a la dinámica del **Framework.Net**. Pero... ¿Qué es el **Framework .Net**? El **Framework .Net** (de ahora en adelante lo llamaré simplemente “**Framework**”) es una extensa colección de **clases** agrupadas en **espacios de nombres** definidos, las cuales están a nuestra disposición para ser usadas desde cualquier lenguaje que lo implemente. El **Framework** está dividido en partes como lo muestra la imagen.

La primera capa, que es la que interactúa directamente con el sistema operativo es la **Common Language Runtime (CLR)**. Esta capa del **Framework** es la encargada de compilar el código **CIL** (anteriormente llamado **MSIL** por **Microsoft Intermediate Language**) a código máquina nativo. El código **CIL** (**Common Intermediate Language**) es el código intermedio que generan los lenguajes de alto nivel establecidos en la **Common Language Specification (CLS)**, la capa más alta del **Framework**. En palabras más simples, podemos concluir que: Desde arriba hacia abajo, el **Framework** está constituido por lenguajes de alto nivel como **C#.Net, Visual Basic .Net,**

**C++ .Net,** etc, que nos permiten escribir código de forma muy agradable.

A diferencia de lenguajes de compilación nativa, como **C, C++** los lenguajes **.Net**, que están definidos por la **CLS**, no compilan su código a código máquina, sino que compilan a un código intermedio llamado **CIL** cuya estructura es bastante similar a la estructura del **Assembler (ASM)**. Esto permite poder trabajar con múltiples lenguajes sobre una misma

solución y que a la hora de compilar todo quede resumido en un mismo tipo de código, el **CIL**, que después será traducido a código ejecutable mediante la **CLR**.



### 3. Todo es un objeto:

A diferencia de muchos otros lenguajes que dicen ser orientados a objetos y en la práctica no resultan serlo, **C#** parte de la base de que **TODO** es un objeto, por tanto implementa al 100% el paradigma

**POO (Programación Orientada a Objetos)**. En este taller todo lo que aprendamos será basado en **POO**, por tanto el estudiante, además de aprender a trabajar con **C#**, aprenderá a pensar los algoritmos y problemas mediante el enfoque tradicional de objetos.

Esta entrega no va a contar con teoría específica sobre objetos, en cambio iremos definiendo y aprendiendo sobre **POO** a medida que los ejemplos vayan implementando sus características. De esta forma no se hace tan tedioso aprender teoría de objetos. De todas maneras, si ustedes lo requieren, en un futuro se puede dedicar una entrega a la teoría de objetos. Recuerden que tienen el foro de **Underc0de** para postear sus consultas y sugerencias.

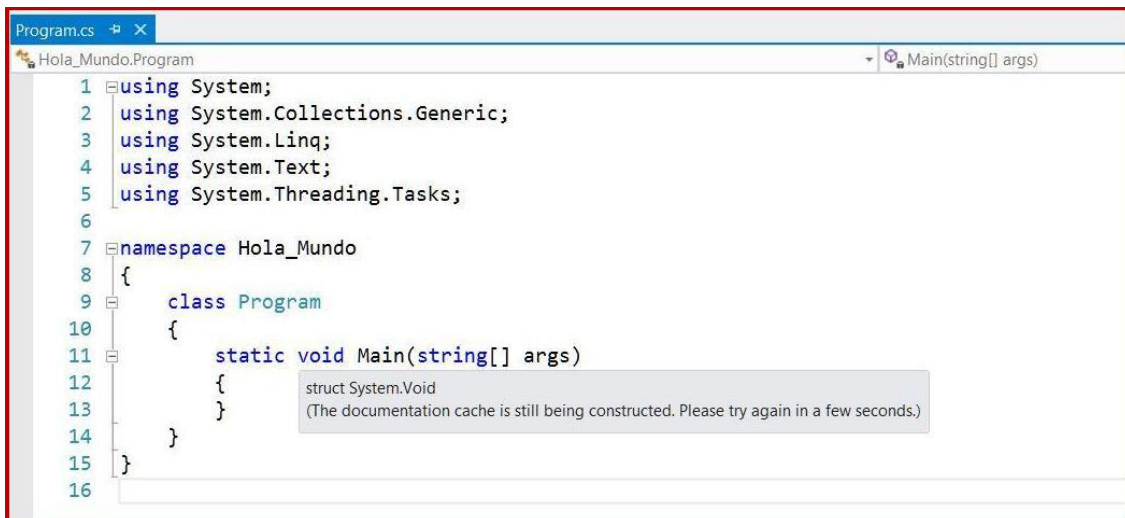
### 4. Esqueleto de un programa en C#:

*Como este no es un taller de programación visual, sólo crearemos proyectos de tipo **consola**. Los proyectos de consola facilitan mucho la legibilidad del código y permiten aprender de forma más clara y ordenada los conceptos.*

Comencemos a analizar detalladamente la estructura que todo programa escrito en **C#** debe cumplir. Para ello abriremos nuestro Visual Studio y crearemos un nuevo proyecto en Visual C# de tipo **consola**.

Tal y como muestra la imagen a continuación, **Visual Studio** nos crea un nuevo proyecto de consola y escribe de forma automática 15 líneas de código. La persona que

conoce **C#** seguro entiende a la perfección el significado de estas líneas, pero vamos a aclararlas, pensando en los que recién están comenzando.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Hola_Mundo
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13        }
14    }
15 }
16
```

Aquí como verán, he creado un nuevo proyecto de consola llamado **Hola Mundo** mediante el cuál vamos a crear el típico hola mundo clásico de todos los lenguajes.

Observemos que las primeras 5 líneas comienzan con la palabra **using**. Esto significa que le estamos diciendo al compilador que vamos a hacer uso de cierto **espacio de nombres** y de todas las clases que existan dentro de él. Por ejemplo, el espacio de nombres **System**, provee al programa de **clases fundamentales** y **clases**

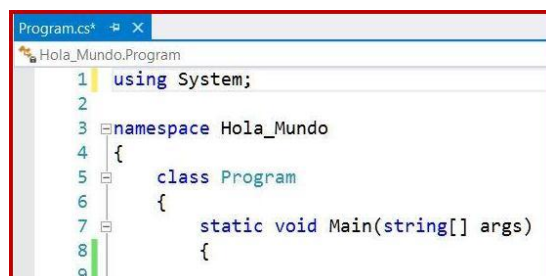
**base** que definen **acciones** y **procedimientos** de uso frecuente, **eventos**, **interfaces** y **excepciones**. Este espacio de nombres es fundamental en nuestros desarrollos.

**System.Collections.Generic:** Este espacio de nombres contiene las clases, interfaces, y otros, necesarias para manejar **colecciones genéricas** de objetos. Esto por ahora no lo vamos a usar, por tanto podemos eliminar esta línea.

**System.Linq:** Este espacio de nombres proporciona clases e interfaces que admiten consultas que utilizan **Language-Integrated Query (LINQ)**. Tampoco lo vamos a usar, por tanto eliminamos esta línea.

**System.Text:** Como su nombre lo indica, este espacio de nombres contiene todo lo necesario para trabajar con **cadenas de caracteres**. Por ahora tampoco lo vamos a usar, por tanto eliminamos la línea.

**System.Threading.Tasks:** Este espacio de nombres proporciona tipos que simplifican la escritura de código **simultáneo** y **asincrónico**. Nada que nos sirva por ahora.



```
1 using System;
2
3 namespace Hola_Mundo
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9
```

Eliminamos también esta línea.

En este punto podemos afirmar que en la cabecera de un programa escrito en C# siempre se deben establecer los espacios de nombres que nuestro programa utilizará. Para esto hacemos uso de la palabra reservada **using**.

Por debajo de los espacios de nombres, tenemos una línea que pone: **namespace Hola\_mundo**. Esto significa que todo lo que esté dentro de las llaves contenedoras “{ }” debe ser tratado como un espacio de nombres nuevo e independiente de los demás. A modo organizacional, esto nos permitiría tener clases de igual nombre pero agrupadas en diferentes **namespaces**. Recordemos que un **namespace** o espacio de nombres es simplemente un **contenedor de clases** que permite tener nuestro programa bien ordenado.

La siguiente línea pone: **class Program** y abre una nueva llave {. Simplemente declara una nueva **clase** llamada **program**. Aquí estarán los **métodos** fundamentales del programa. ¿Pero que es todo esto de **clases** y **métodos**? ¿Recuerdan que hace un rato dijimos que para C# absolutamente **TODO es un objeto**? Bueno, ésta es la primera prueba de ello. Viendo este panorama, entendemos que el programa se trata a sí mismo como un objeto. Dijimos también que comenzaríamos abordando la teoría de objetos según vayan apareciendo sus elementos en los ejemplos. Bueno, aquí tenemos los primeros dos: **Clase** y **Método**.

Una clase no es más que un simple “**molde**” en base al cuál, posteriormente, se creará un **objeto**. Un

**objeto**, para ser **objeto**, tiene que **construirse** en base a una **clase**. Por ejemplo, yo puedo construir un objeto

“**empleado**” basándome en la clase “**empleados**”.

Un **método** es el conjunto de **funcionalidad** que una **clase** define a efectos de que el objeto realice una acción en concreto. Tal vez los mareé un poco con la definición, pero descuiden que nada es tan trágico. Supongamos que creamos una clase llamada “**Televisores**”. Esta clase debe definir el comportamiento que tendrán los objetos que de ella nazcan. Entonces la pregunta que el programador debe hacerse es: ¿Qué acciones realizan los televisores? Cada respuesta acertada será un nuevo método de la clase. Por ejemplo: **Encender, Apagar, SubirVolumen, BajarVolumen, SubirCanal, BajarCanal**, etc.

Mediante los métodos públicos es que el objeto puede ser “**controlado**” desde afuera, por ejemplo desde otro **objeto**. Para la gente que tiene experiencia en lenguajes procedurales, los métodos son lo que en dichos lenguajes se llamaban **Funciones** y **Procedimientos**.

Además de los métodos, las clases constan de otras partes que iremos descubriendo más adelante.

La siguiente línea pone: **static void Main(string[] args)**. Esto es la declaración del método **Main**. Este método es el **punto de entrada** al programa. Cuando compilamos, el compilador buscará entre los archivos de código de fuente éste método, y cuando lo encuentre sabrá que es por aquí por donde debe comenzar a ejecutar el

*El punto de entrada a un programa escrito en C# siempre es el método MAIN().*

programa. La palabra **static** indica que el método es estático, por tanto puede llamarse desde la clase, no hace

falta tener un objeto creado para utilizar este método, simplemente invocando la

clase a la cual pertenece es suficiente para usarlo. La palabra **void** indica que es un

método que no espera retorno de valores. Esto lo veremos en detalle más adelante.

## 5. La clase Console:

Estamos creando un proyecto de **consola**, por tanto es obligatorio poder interactuar de forma clara y directa con ésta. Para esto el **Framework** incluye, dentro del **namespace System** una clase llamada **Console**. Esta clase tiene como particularidad que no puede ser **heredada**. Esto lo comento de pasada ya que por ahora no entraré en tema **herencia**.

La clase **Console** nos brinda un montón de métodos mediante los cuales podemos interactuar con la **consola del sistema**. Todos estos métodos son **estáticos**, o sea que para usarlos basta con llamar a su clase.

Por el momento vamos a aprender 4 métodos fundamentales de esta clase:

### - **Console.WriteLine();**

Este método sirve para escribir texto en la consola y luego de escribirlo hacer un salto de línea. El método requiere como **parámetro** un valor de tipo **cadena** que será el que escribirá en la consola. Por ejemplo: **Console.WriteLine("Hola Mundo");** Esta línea escribirá la frase Hola Mundo en la consola y luego hará un salto de línea.

### - **Console.Write();**

Este método realiza lo mismo que el anterior pero sin hacer un salto de línea al final.

### - **Console.ReadLine();**

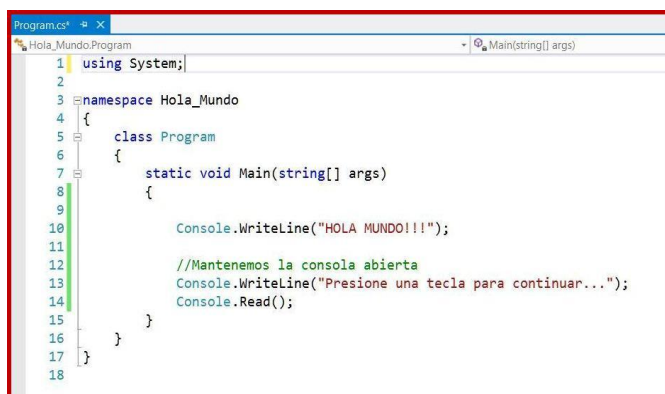
Este método sirve para leer y almacenar en una variable lo que el usuario escriba en la consola. Si queremos almacenar la entrada del usuario, debemos asignar el resultado del método a una variable de tipo **string**, sino, el programa se detendrá hasta que el usuario presione una tecla. Luego de leer lo ingresado por el usuario, realiza un salto de línea en la consola.

### - **Console.Read();**

Lo mismo que el método anterior pero sin salto de línea al final.

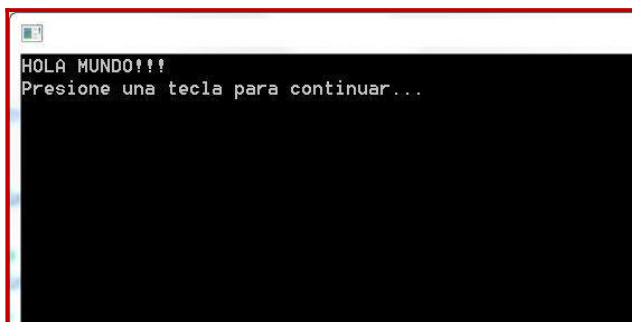
## 6. Nuestro Primer Programa en C#:

Ahora que sabemos todo esto podemos echar mano al código que venimos escribiendo y agregarle estas líneas al método **main()**;



```
1 using System;
2
3 namespace Hola_Mundo
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9
10            Console.WriteLine("HOLA MUNDO!!!");
11
12            //Mantenemos la consola abierta
13            Console.WriteLine("Presione una tecla para continuar...");
14            Console.Read();
15        }
16    }
17 }
18
```

Si todo está correcto, nuestro programa **Hola Mundo** debería verse tal y como el de la imagen.



Presionamos la tecla **F5** para compilar y ejecutar el proyecto y deberíamos tener frente a nuestros ojos la ventana de consola del sistema operativo mostrando la leyenda “**Hola Mundo**” y esperando que el usuario presione una tecla para continuar.

```
Programas - x
% Constantes.Program - | Main(string[] args)
1 using System;
2
3 namespace Constantes
4 {
5     class Program
6     {
7
8         const double PI = 3.14;
9
10        static void Main(string[] args)
11        {
12            float radio = 0;
13
14            Console.Write("Ingrese radio: ");
15            radio = float.Parse(Console.ReadLine());
16            Console.WriteLine("El area es: " + (PI * radio * radio));
17            Console.ReadLine();
18        }
19    }
20 }
21
```

Hemos entonces programado y creado efectivamente nuestro primer programa en un lenguaje totalmente orientado a objetos como es **C# .Net**.

También hemos aprendido o repasado un poco sobre la teoría de objetos

Basándonos en el paradigma **POO**. Por supuesto que esto no acaba acá. La idea de este taller es simplemente encender en ustedes la llama de la curiosidad para que luego salgan a buscar información en internet y enriquezcan así los conocimientos.

Ésta fue una entrega más que nada introductoria al lenguaje y al paradigma **POO** con la intención de nivelar a los lectores que recién inician en el tema y que puedan de esta manera entender mejor las entregas que siguen en donde el nivel ira aumentando progresivamente.

## 7. Variables y Constantes:

Hemos alcanzado un punto en el que medianamente ya nos damos cuenta qué es y cómo funciona un programa en **C# .Net**. Ahora es momento de aprender los primeros elementos del lenguaje como son las **variables** y las **constantes**.

### 8. Constantes:

Una constante es un valor que se conoce en tiempo de compilación, y que no cambiará jamás a lo largo de toda la ejecución del programa. Para ser más claros, citemos un ejemplo. Muchas veces en programación, tal y como pasa en el mundo de las matemáticas, nos vamos a ver en la necesidad de trabajar con valores que son constantes, como por ejemplo el número **PI** en un programa que calcule áreas de circunferencias. El valor de **PI** es un valor constante: 3.14... Comprobaremos que es mucho más simple y nos ahorrará mucho tiempo trabajar con una constante **PI** definida al principio del programa, cuyo valor sea 3.14.



Es posible que nuestro programa tenga múltiples cálculos que hagan uso del valor de **pi**. Ahora... ¿Qué pasaría si de pronto nos damos cuenta de que necesitamos una mayor precisión de cálculo, y en vez de considerar **PI** = 3.14 necesitamos considerar **PI** = 3.14159265358979323846? Sería muy engorroso tener que recorrer todo el código y reemplazar uno a uno los valores. Mucho más simple en cambio es reemplazar el valor de **PI** en la constante definida al principio del programa o método.

Para declarar una constante en **C#** necesitamos usar la palabra reservada **const** seguida del tipo de dato de la constante. En este caso, como **PI** es un valor de coma flotante con varias posiciones decimales, usaremos el tipo de dato **double**. La declaración entonces quedará así: **const double PI = 3.14;**

## 9. Variables:

Veamos ahora qué son y cómo se usan las variables en **C#**. Una variable no es más que un espacio en la memoria atado a un identificador. Hablando un poco más claramente, podemos decir que una variable es un tipo de dato identificado dentro de un ámbito de ejecución, cuyo contenido puede variar. Una vez que la variable se encuentre declarada, su contenido puede ser reasignado tantas veces como se desee.

**C#** es un lenguaje **fuertemente tipado**, es decir que todo valor, variable o constante debe obedecer si o si a algún tipo de dato. Según lo establece la **CLS (common language specification)**, **C#** admite dos tipos de datos: **Tipos de datos integrados** y **Tipos de datos personalizados**.

Los tipos de datos integrados están definidos por el Framework y son los siguientes:

| <i>The Intrinsic Data Types of C#</i> |                       |                    |   |  |
|---------------------------------------|-----------------------|--------------------|---|--|
| <b>C# Shorthand</b>                   | <b>CLS Compliant?</b> | <b>System Type</b> | <b>Range</b>  | <b>Meaning in Life</b>                           |
| bool                                  | Yes                   | System.Boolean     | true or false   | Represents truth or falsity                      |
| sbyte                                 | No                    | System.SByte       | -128 to 127   | Signed 8-bit number                              |
| byte                                  | Yes                   | System.Byte        | 0 to 255  | Unsigned 8-bit number                            |
| short                                 | Yes                   | System.Int16       | -32,768 to 32,767                                       | Signed 16-bit number                             |
| ushort                                | No                    | System.UInt16      | 0 to 65,535   | Unsigned 16-bit number                           |
| int                                   | Yes                   | System.Int32       | -2,147,483,648 to 2,147,483,647                         | Signed 32-bit number                             |
| uint                                  | No                    | System.UInt32      | 0 to 4,294,967,295                                      | Unsigned 32-bit number                           |
| long                                  | Yes                   | System.Int64       | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | Signed 64-bit number                             |
| ulong                                 | No                    | System.UInt64      | 0 to 18,446,744,073,709,551,615                         | Unsigned 64-bit number                           |
| char                                  | Yes                   | System.Char        | U+0000 to U+ffff  | Single 16-bit Unicode character                  |
| float                                 | Yes                   | System.Single      | $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$   | 32-bit floating-point number                     |
| double                                | Yes                   | System.Double      | $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ | 64-bit floating-point number                     |
| decimal                               | Yes                   | System.Decimal     | $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$   | 96-bit signed number                             |
| string                                | Yes                   | System.String      | Limited by system memory                                | Represents a set of Unicode characters           |
| Object                                | Yes                   | System.Object      | Can store any data type in an object variable           | The base class of all types in the .NET universe |

Los tipos de datos personalizados son los siguientes: **struct**, **class**, **interface** y **enum**. Se llaman personalizados porque es el programador quien determina la construcción de cada uno de estos tipos de datos. Veremos cada uno de ellos en profundidad más adelante en el taller.

Como podrán apreciar en la imagen anterior, tenemos aquí un simple programa que declara una variable llamada **Nombre**, y luego le asigna el resultado de una entrada de consola hecha por el usuario para finalmente mostrar el contenido concatenado a una inscripción.

Como habrán notado, para declarar una variable, basta con anteponer el tipo de dato que se desee, al nombre de la variable. No olvidemos finalizar la declaración con un “;”.

En este punto quiero hacer una pequeña acotación. El tema de los tipos de datos es muy amplio y tiene muchos campos en donde se podría profundizar. Como esto no es un curso de .Net Framework ni es un manual de referencia del lenguaje, ni pretende ser un tratado de desarrollo de software mediante **C# .net**, sino que esto es un taller de acercamiento al lenguaje y un “Jump Start” para quienes están pensando en dar sus primeros pasos en el mundo de la programación, es que he optado por tocar este tema sin entrar en la profundidad que el tema merecería. Sin embargo, para quienes quieran ahondar más en esta materia, les dejo unos links con documentación detallada al respecto:

- 1) <http://www.ehu.es/mrodriguez/archivos/csharp/pdf/Lenguaje/Tipos.pdf>
- 2) [http://msdn.microsoft.com/es-es/library/ms173104\(v=vs.110\).aspx](http://msdn.microsoft.com/es-es/library/ms173104(v=vs.110).aspx)
- 3) [http://es.wikipedia.org/wiki/C\\_Sharp#Tipos\\_de\\_datos](http://es.wikipedia.org/wiki/C_Sharp#Tipos_de_datos)

## 10. Inicialización de variables:

Una variable puede declararse optando por inicializarla en algún valor, o no. Si la variable no está inicializada, no se podrá consultar su contenido hasta que se le asigne un valor ya que si lo hacemos obtendremos una excepción. Es buena práctica de programación siempre inicializar las variables, aunque sea en valores como “0”, “”, y **null**.

**A no confundir: Una variable inicializada en null no es lo mismo que si no estuviera inicializada.**

```
string Nombre = null;  
string Apellido = "";  
int edad = 0;
```

Acá tenemos tres variables. Nombre está inicializada en **null**, Apellido en vacío (“”) y edad en **0**. La línea punteada verde indica que esas variables aún no han sido utilizadas en ninguna parte del código fuente anterior o posterior. ¡El IDE de Visual Studio es muy inteligente!

```
using System;  
  
namespace Variables  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Declaramos una variable de tipo string llamada Nombre.  
            string Nombre;  
  
            Console.Write("Por favor, ingrese su nombre: ");  
  
            // Asignamos a la variable nombre lo que el usuario  
            // ha ingresado mediante la consola.  
            Nombre = Console.ReadLine();  
  
            // Limpiamos la consola  
            Console.Clear();  
  
            //Mostramos el contenido de la variable Nombre  
            Console.WriteLine("Su nombre es {0}\n\n", Nombre);  
  
            Console.Write("Presione una tecla para continuar... ");  
            Console.Read();  
        }  
    }  
}
```

## 11. El operador de asignación “=”:

Hemos hablado ya de declaración, tipo de dato e inicialización de variables, pero aún no hemos dicho nada sobre cómo hay que proceder para asignar un valor a una variable ni tampoco hemos hablado sobre cómo se compara o se consulta el valor de una variable.

Para asignar un valor a una variable necesitamos dos cosas:

- 1) Que el tipo de dato del valor y el tipo de dato de la variable sean iguales o compatibles ya que si tratáramos, por ejemplo de asignar el valor “**Barcelona**” a la variable **Edad** de tipo entero (**int**) de seguro obtendríamos un error de compilación.
- 2) Hacer uso del operador de asignación representado por el signo “igual” (=).

Entonces si escribimos:

```
int edad;  
edad = 30;
```

Estaríamos declarando la variable edad y en la siguiente línea le asignamos el valor 30. **C#** también nos permite declarar e inicializar en la misma línea. Por tanto:

```
int edad = 30;
```

La línea anterior es completamente válida.

## 12. Operadores Relacionales:

Muchas veces nos veremos en la necesidad inminente de comparar si el valor de una variable es igual, o si es mayor, o menor que el valor de otra variable, o constante o literal. Para estos fines, el lenguaje acepta el uso de operadores de comparación llamados **operadores relacionales**. Estos son:

| Operador | Significado                        |
|----------|------------------------------------|
| ==       | Igual que                          |
| >        | Mayor que                          |
| <        | Menor que                          |
| >=       | Mayor o igual que                  |
| <=       | Menor o igual que                  |
| !=       | No es igual que o es diferente que |

Siempre que queramos relacionar un valor con otro, debemos hacer uso de estos operadores de relación. Gracias a ellos podremos determinar cuál valor es mayor, menor, igual o desigual. Volveremos a analizar este tema más en detalle cuando estudiemos estructuras condicionales en futuras entregas.

## 13. Ejercicio de aplicación:

- 1- Pongamos a prueba lo aprendido. Este ejercicio consiste en desarrollar un programa de consola en **C#** que al iniciar pida al usuario los siguientes datos:

- 1) **Apellido**
- 2) **Nombre**
- 3) **Edad**
- 4) **País**
- 5) **Altura**
- 6) **Peso**
- 7) **Email**

Luego, el programa debe limpiar la pantalla y mostrar toda esta información a modo de ficha. Cualquier duda o pregunta que surja, no duden en postear en el foro de **underc0de** en el apartado de programación.

- 2- Escribir un programa en **C#** que permita calcular el promedio de 5 calificaciones obtenidas por un alumno. El programa debe solicitar el apellido y nombre del alumno y las calificaciones obtenidas en: Lengua, Matemática, Historia, Física y Geografía. Luego debe calcular e imprimir en pantalla el promedio obtenido.  
**TIP: Para este ejercicio se deben usar operadores matemáticos (+, -, \*, /) y variables de tipo float.**

**This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>.**