# Introducción a C#

## CONTENIDO

## ➢ ESTRUCTURA DE UN PROGRAMA C#

### HOLA, MUNDO

```
using System;

class Hola
{
  public static void Main()
  {
    Console.WriteLine("Hola, mundo");
  }
}
```

**Explicación previa**

El primer programa que se suele escribir en un lenguaje nuevo es el inevitable Hola, mundo.

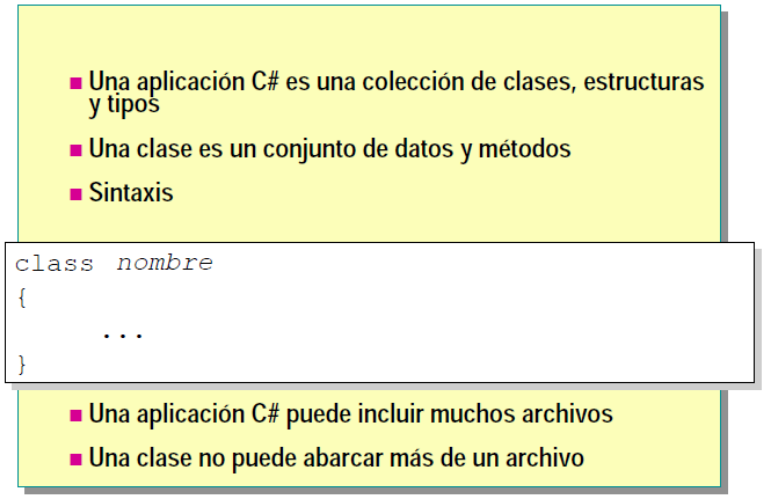
El primer programa que se suele escribir cuando se aprende un lenguaje nuevo es el inevitable Hola, mundo. En este módulo tendrá la oportunidad de examinar la versión en C# de este primer programa.

El código de ejemplo en la transparencia contiene todos los elementos esenciales de un programa C# y es muy fácil de probar. Lo único que hace cuando se ejecuta desde la línea de comandos es mostrar el siguiente mensaje:

```
Hola, mundo
```

En los temas que siguen analizaremos este programa simple para aprender más sobre los componentes de un programa C#.

## La Clase



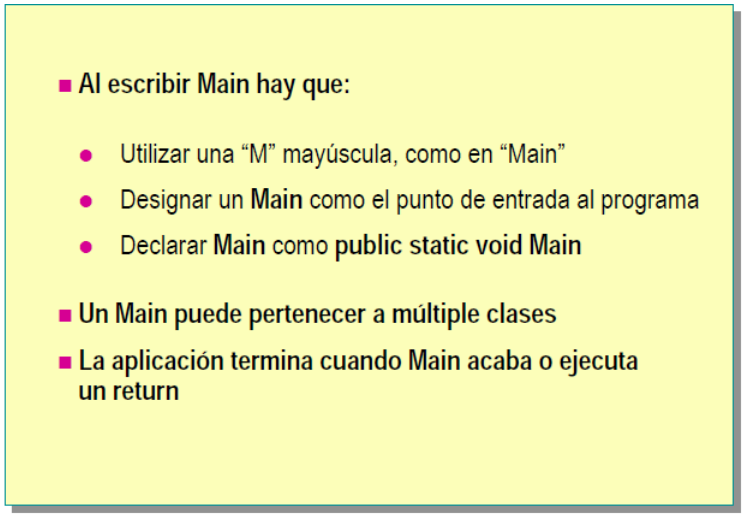

**Explicación previa**

Una aplicación C# es una colección de una o más clases.

Un examen del código para la aplicación Hola, mundo revela que hay una sola clase llamada **Hola**. Esta clase se introduce con la palabra clave **class**. Después del nombre de la clase hay una llave de apertura ({). Todo lo que hay hasta la correspondiente llave de cierre (}) forma parte de la clase.

Las clases para una aplicación C# se pueden extender a uno o más archivos. Es posible poner varias clases en un archivo, pero una sola clase no puede abarcar más de un archivo.

## El método Main

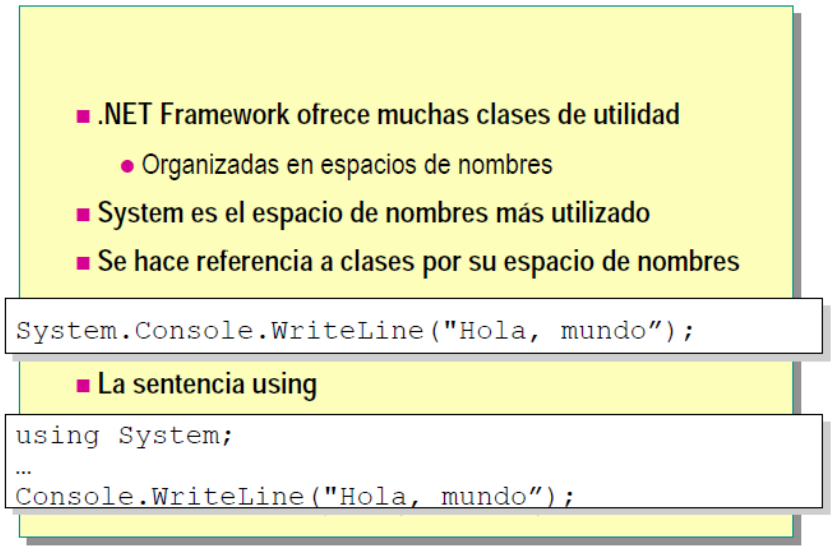


El lenguaje C# distingue entre mayúsculas y minúsculas. **Main** debe estar escrito con una "M" mayúscula y con las demás letras en minúsculas.

También es importante la firma de **Main**. Si se emplea Visual Studio, se creará automáticamente como **static void** (como veremos más adelante en el curso). No se debe cambiar la firma si no hay una buena razón para hacerlo.

La aplicación se ejecuta hasta llegar al final de **Main** o hasta que **Main** ejecuta una instrucción **return**.

## LA SENTENCIA USING Y EL ESPACIO DE NOMBRES SYSTEM




Como parte de Microsoft .NET Framework, C# incluye muchas clases de utilidad que realizan una gran variedad de operaciones útiles. Estas clases están organizadas en *espacios de nombres*, que son conjuntos de clases relacionadas. Un espacio de nombres también puede contener otros espacios de nombres.

.NET Framework está compuesto por muchos espacios de nombres, el más importante de los cuales se llama **System**. El nombre de espacios **System** contiene las clases que emplean la mayor parte de las aplicaciones para interactuar con el sistema operativo. Las clases más utilizadas son las de entrada y salida (E/S). Como ocurre en muchos otros lenguajes, C# no tiene funciones propias de E/S y por tanto depende del sistema operativo para ofrecer una interfaz compatible con C#.

Para hacer referencia a objetos en espacios de nombres se utiliza un prefijo explícito con el identificador del espacio de nombres. Por ejemplo, el espacio de nombres **System** contiene la clase **Console**, que a su vez contiene varios métodos, como **WriteLine**. Se puede acceder al método **WriteLine** de la clase **Console** de la siguiente manera:

```
System.Console.WriteLine("Hola, mundo");
```

Sin embargo, el uso de un nombre completo para referirse a objetos puede resultar poco manejable y propicio a errores. Para hacerlo más sencillo, se puede especificar un espacio de nombres poniendo una sentencia **using** al comienzo de la aplicación, antes de la definición de la primera clase. Una sentencia **using** especifica el espacio de nombres que se examinara si una clase no está definida

explícitamente en la aplicación. Es posible poner más de una sentencia **using** en el archivo de origen, pero todas tienen que ir al principio del archivo.

Con la sentencia **using** se puede escribir el código anterior de la siguiente manera:

```
using System;
...
Console.WriteLine("Hola, mundo");
```

En la aplicación Hola, mundo, la clase **Console** no está definida explícitamente. Cuando se compila la aplicación, el compilador busca **Console** en el espacio de nombres **System** y genera código que hace referencia al nombre completo **System.Console**.

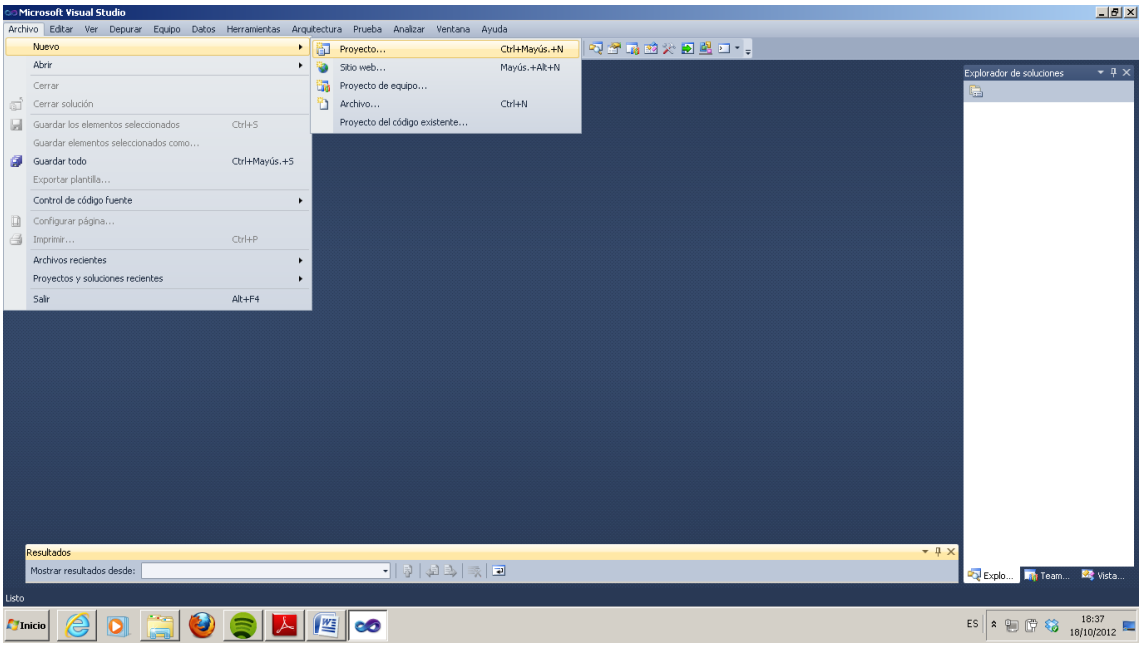
## USO DE VISUAL STUDIO PARA CREAR UN PROGRAMA C#

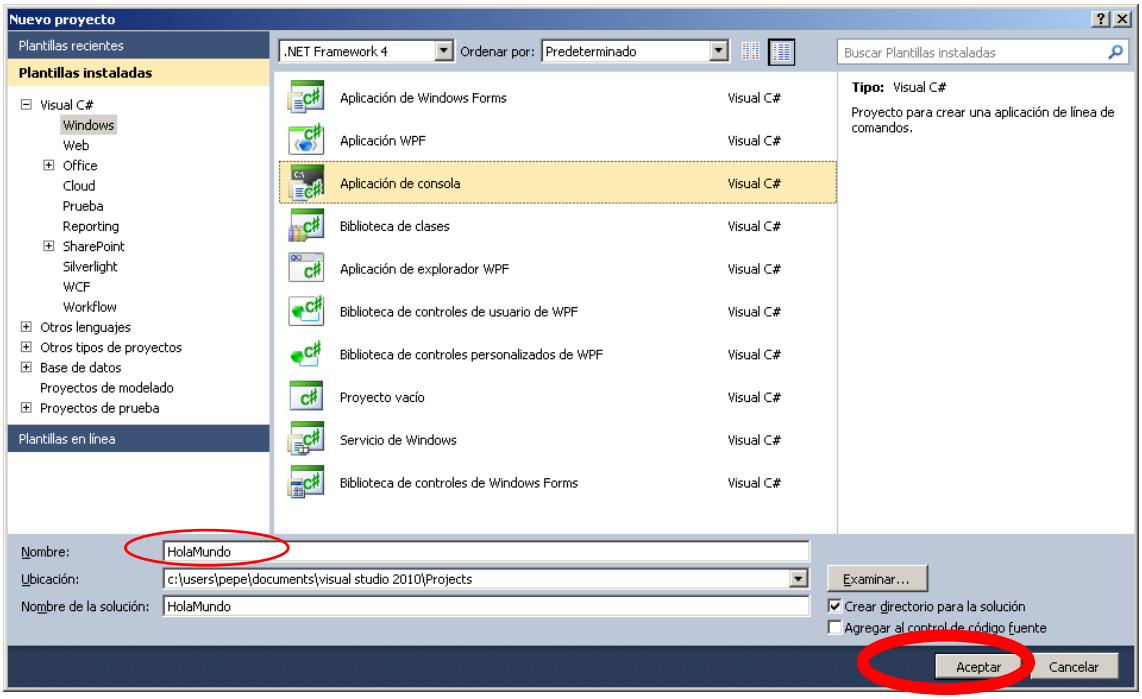
Cómo crear una aplicación C#. Archivo.Nuevo.Proyecto. Se abrirá una ventana y seleccionamos aplicación de consola. En la parte inferior nos pedirá Nombre de la aplicación, ubicación y nombre de la solución. Para nosotros por ahora serán los mismos nombres. Le damos aceptar y se no abrirá el código fuente de nuestro primer programa "program.cs". Con un código como el que sigue:

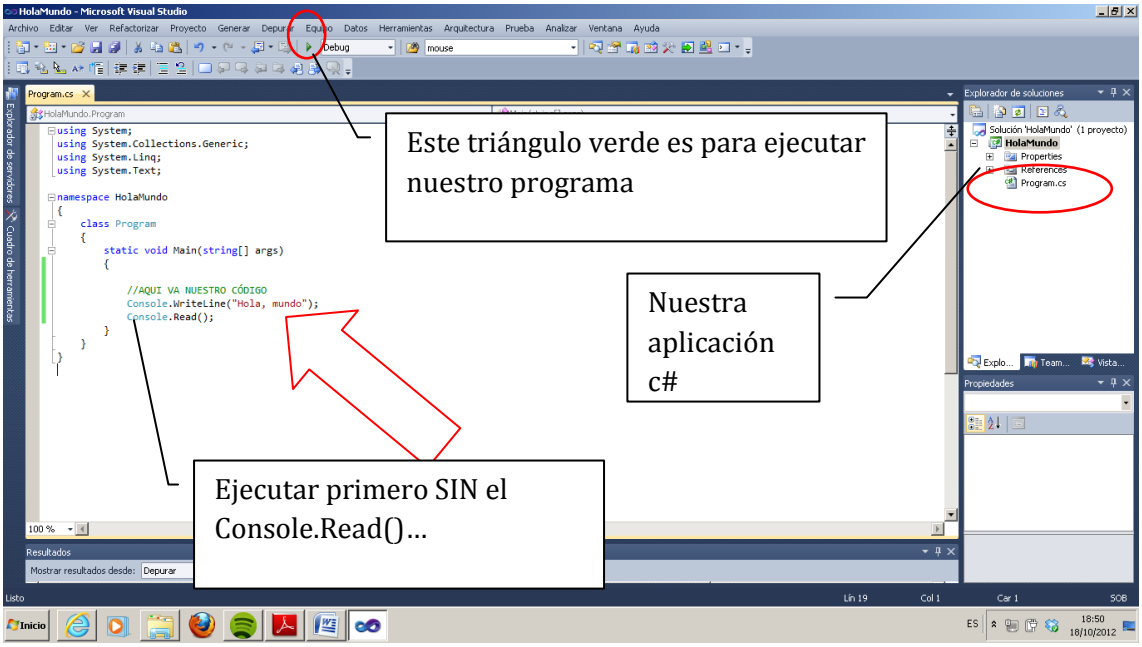
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

```
…y en la parte de la derecha está la estructura de la solución que consta de:
Nivel 1 nombre de la solución, una solución puede tener varios proyectos.
nivel 2 nombre del proyecto
nivel 3 carpeta properties que contiene información interna del visual studio
nivel 4 la carpeta references que contiene las librerías o using o espacios de
nombre que necesitará el programa.
Y por último el código fuente de nuestra aplicación el program.cs
Vayamos a ejecutarlo pero antes escribiremos nuestra primera línea de código
saludando al mundo entre las llaves del apartado        static void Main(string[]
args) para ello escribimos lo siguiente:
Console.WriteLine("Hola Mundo...");

Pulsamos control+F5 y nos saldrá la consola de Windows con el texto hola mundo y
otro texto que pondrá pulse tecla para continuar…

También podremos ejecutarlo desde la opción de menú Depurar.Iniciar sin depurar
La otra opción Iniciar depuración o F5 se verá cuando utilizemos la depuración o
Debug para los errores.
```

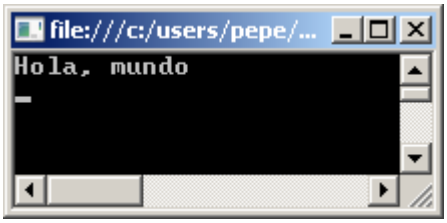
Como se muestra en la figura.


Microsoft Visual Studio
Archivo Editar Ver Depurar Equipo Datos Herramientas Arquitectura Prueba Analizar Ventana Ayuda
Nuevo
Abrir
Cerrar
Cerrar solución
Guardar los elementos seleccionados Ctrl+S
Guardar elementos seleccionados como...
Guardar todo Ctrl+Mayús.+S
Exportar plantilla...
Control de código fuente
Configurar página...
Imprimir... Ctrl+P
Archivos recientes
Proyectos y soluciones recientes
Salir Alt+F4
Proyecto... Ctrl+Mayús.+N
Sitio web... Mayús.+Alt+N
Proyecto de equipo...
Archivo... Ctrl+N
Proyecto del código existente...
Explorador de soluciones
Resultados
Mostrar resultados desde:
Listo
Inicio
ES
18:37
18/10/2012


Nuevo proyecto
Plantillas recientes
Plantillas instaladas
Visual C#
Windows
Web
Office
Cloud
Prueba
Reporting
SharePoint
Silverlight
WCF
Workflow
Otros lenguajes
Otros tipos de proyectos
Base de datos
Proyectos de modelado
Proyectos de prueba
Plantillas en línea
.NET Framework 4
Ordenar por: Predeterminado
Buscar Plantillas instaladas
Aplicación de Windows Forms Visual C#
Aplicación WPF Visual C#
Aplicación de consola Visual C#
Biblioteca de clases Visual C#
Aplicación de explorador WPF Visual C#
Biblioteca de controles de usuario de WPF Visual C#
Biblioteca de controles personalizados de WPF Visual C#
Proyecto vacío Visual C#
Servicio de Windows Visual C#
Biblioteca de controles de Windows Forms Visual C#
Tipo: Visual C#
Proyecto para crear una aplicación de línea de comandos.
Nombre: HolaMundo
Ubicación: c:\users\pepe\documents\visual studio 2010\Projects
Nombre de la solución: HolaMundo
Examinar...
Crear directorio para la solución
Agregar al control de código fuente
Aceptar
Cancelar




F5 o el triangulo verde y se ejecutará... Se puede omitir el console.Read(), pero si queremos ver el resultado tendremos que ejecutarlo con control+F5




# ➢ OPERACIONES BÁSICAS DE ENTRADA/SALIDA

## LA CLASE CONSOLE

La clase **Console** hace que una aplicación C# pueda acceder a las secuencias estándar de entrada, salida y error.

La entrada estándar está asociada normalmente con el teclado, de forma que todo lo que el usuario escribe en el teclado se puede leer desde la secuencia de entrada estándar. Del mismo modo, la secuencia de salida estándar suele estar dirigida a la pantalla, al igual que la secuencia de error estándar.

**Nota**

Estas secuencias y la clase **Console** sólo tienen sentido para aplicaciones de consola, que son las que se ejecutan en una ventana Command (Comandos).

Es posible direccionar cualquiera de las tres secuencias (entrada estándar, salida estándar, error estándar) a un archivo o dispositivo, tanto durante la programación como al ejecutar la aplicación.

# LOS MÉTODOS WRITE Y WRITELINE

Los métodos **Console.Write** y **Console.WriteLine** se pueden utilizar para mostrar información en la pantalla de la consola. Los dos métodos son muy similares; la diferencia más importante es que **WriteLine** añade un fin de línea/retorno de carro al final de la salida, mientras que **Write** no lo hace.

Ambos métodos son sobrecargados. Puede explicarlo con números variables y tipos de parámetros. Por ejemplo, puede usar el siguiente código para escribir "99" en la pantalla:

```
Console.WriteLine(99);
```

También puede usar el siguiente código para escribir "Hola, mundo" en la pantalla:

```
Console.WriteLine("Hola, mundo");
```

## Formatos de texto

Existen formas de **Write** y **WriteLine** más potentes, que toman una cadena de formato y parámetros adicionales. La cadena de formato especifica cómo aparecen los datos y puede contener marcadores, que son sustituidos por los parámetros que siguen. Por ejemplo, puede usar el siguiente código para mostrar el mensaje "La suma de 100 y 130 es 230":

```
Console.WriteLine("La suma de {0} y {1} es {2}", 100, 130, 100+130);
```

El primer parámetro que sigue a la cadena de formato se referencia como parámetro cero: {0}.

Se puede utilizar el parámetro de la cadena de formato para especificar anchuras de campos y para indicar si los valores en esos campos deben ir justificados a derecha o a izquierda, como se ve en el siguiente código:

```
Console.WriteLine("\"Justificación a la izquierda de un campo de anchura 10: {0,-10}\"",99);
Console.WriteLine("\"Justificación a la derecha de un campo de anchura 10: {0,10}\"", 99);
```

Para desactivar el significado especial de un carácter en una cadena de formato se puede usar una barra diagonal inversa (\) antes de ese carácter. Por ejemplo, "\{" hará que aparezca un "{" literal y "\\" mostrará un "\" literal. También se puede emplear el carácter @ para representar al pie de la letra una cadena entera. Por ejemplo, @"\\server\share" dará como resultado \\server\share.

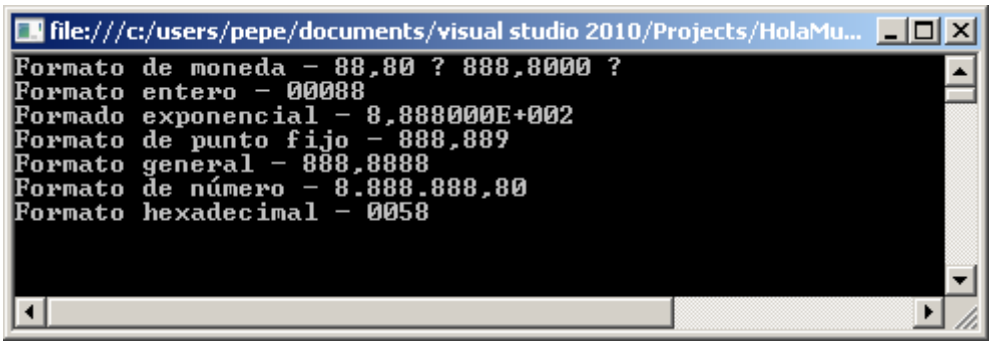
## Formatos numéricos

Es posible utilizar la cadena de formato para especificar el formato de datos numéricos. La sintaxis completa para la cadena de formato es `{N,M: FormatString}`, donde `N` es el número del parámetro, `M` es la anchura y justificación del campo, y `FormatString` indica cómo se deben mostrar los datos numéricos. La tabla siguiente resume los valores que puede adoptar **FormatString**. Opcionalmente, en todos estos formatos se puede especificar el número de dígitos que se desea mostrar o al que se debe redondear.

| Valor | Significado |
|---|---|
| C | Muestra el número como una unidad monetaria, usando el símbolo y las convenciones de la moneda local. |
| D | Muestra el número como un entero decimal. |
| E | Muestra el número como usando notación exponencial (científica). |
| F | Muestra el número como un valor en coma fija. |
| G | Muestra el número como un valor entero o en coma fija, dependiendo del formato que sea más compacto. |
| N | Muestra el número con comas incorporadas. |
| X | Muestra el número utilizando notación hexadecimal. |

```
Console.WriteLine("Formato de moneda - {0:C} {1:C4}", 88.8, 888.8);
Console.WriteLine("Formato entero - {0:D5}", 88);
Console.WriteLine("Formado exponencial - {0:E}", 888.8);
Console.WriteLine("Formato de punto fijo - {0:F3}", 888.8888);
Console.WriteLine("Formato general - {0:G}", 888.8888);
Console.WriteLine("Formato de número - {0:N}", 8888888.8);
Console.WriteLine("Formato hexadecimal - {0:X4}", 88);
```

| | | |
|---|---|---|
| | | Console.Write("{0:C}", 2,5); |
| C o c | Moneda | |
| | | Console.Write("{0:C}", -2,5); |
| D o d | Decimal | Console.Write("{0:D5}", 25); |
| E o e | Científico | Console.Write("{0:E}", 250000); |
| | | Console.Write("{0:F2}", 25); |
| F o f | Punto fijo | |
| | | Console.Write("{0:F0}", 25); |
| G o g | General | Console.Write("{0:G}", 2,5); |
| N o n | Número | Console.Write("{0:N}", 2500000); |
| | | Console.Write("{0:X}", 250); |
| X o x | Hexadecimal | |
| | | Console.Write("{0:X}", 0xffff); |



# LOS MÉTODOS READ Y READLINE

Los métodos **Console.Read** y **Console.ReadLine** se pueden utilizar para mostrar información introducida por el usuario con el teclado.

## El método Read

**Read** lee el siguiente carácter desde el teclado. Devuelve el valor **int** –1 si ya no hay más información. De lo contrario, devuelve un **int** que representa el carácter leído.

## El método ReadLine

**ReadLine** lee todos los caracteres hasta el final de la línea introducida (el retorno de carácter de carro). La información introducida se devuelve como una cadena de caracteres. El siguiente código se puede utilizar para leer una línea de texto desde el teclado y mostrarla en la pantalla:

```
string input = Console.ReadLine();
Console.WriteLine("{0}", input);
```


### COMENTARIOS A APLICACIONES

Es importante que todas las aplicaciones tengan una documentación adecuada. Siempre hay que incluir suficientes comentarios para que un desarrollador que no participara en la creación de la aplicación original pueda seguir y comprender su funcionamiento. Los comentarios deben ser exhaustivos y pertinentes. Unos buenos comentarios añaden información que no es fácil de expresar usando sólo las instrucciones del código, ya que explican el "porqué" en lugar del "qué." Siga las normas de su organización para comentar código (si las tiene).

C# ofrece varios mecanismos que permiten añadir comentarios al código de aplicaciones: comentarios de una sola línea, comentarios de varias líneas y documentación generada en XML.

Para añadir un comentario de una sola línea se pueden utilizar los caracteres de barra diagonal (//). Al ejecutar la aplicación, se ignora todo lo que sigue a estos dos caracteres hasta el final de la línea.

También es posible hacer comentarios en bloques de varias líneas. Un comentario en bloque empieza con el par de caracteres /* y continúa hasta llegar al el par de caracteres */ correspondiente. Los comentarios en bloques no pueden estar anidados.

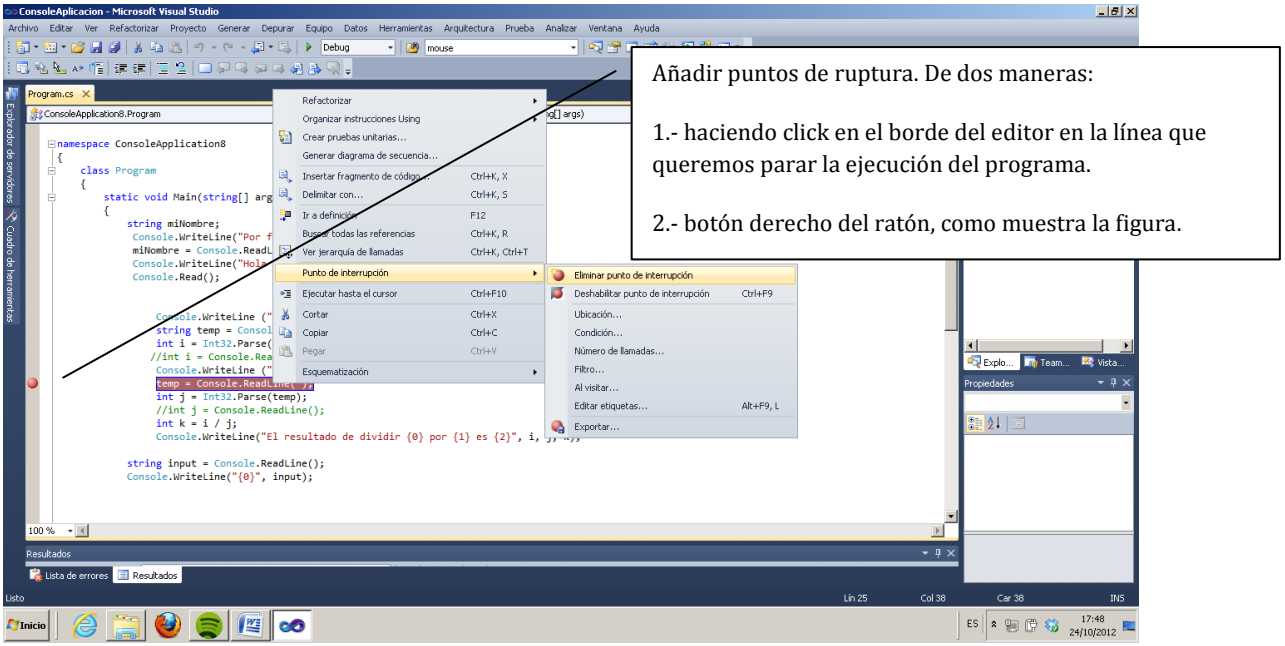
Para comentar funciones podemos poner /// y el nos complementa como se muestra

```
/// <summary>
///
 /// </summary>
 /// <param name="args"></param>
```

# ➢ COMPILACIÓN, EJECUCIÓN Y DEPURACIÓN

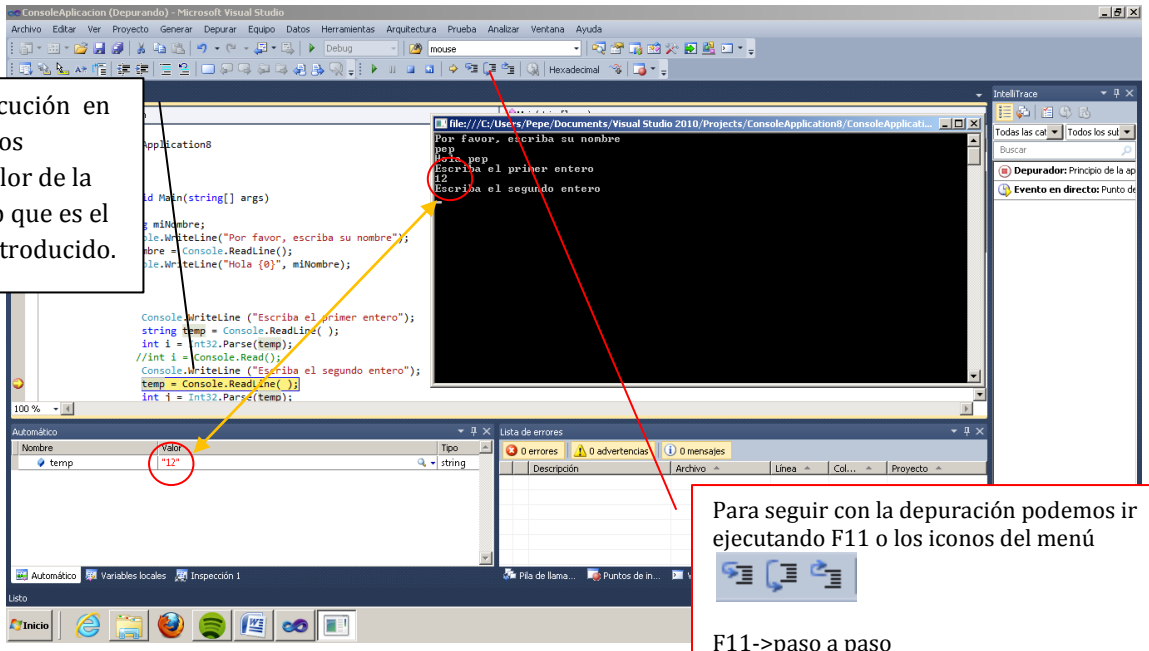
Cuando empecemos a programar habrán situaciones que desconozcamos el motivo por el cual falla, es en ese momento cuando tendremos que utilizar el debugger y depurar la aplicación para ver el estado de las variables en cada momento.

Para ello haremos: colocaremos un breakpoint sobre la línea que queremos analizar, para ello en el código fuente y hacemos click en el margen izquierdo del código fuente a la altura de la línea o bien sobre la línea hacemos pulsamos el botón derecho del ratón nos saldrá un menú emergente vamos a la opción punto de interrupción y seleccionamos insertar punto de interrupción.




Ejecutamos con F5 y en la parte inferior izquierda del Visual Studio nos sale una ventana con el valor de todas las variables en ese momento y pulsando F10 vamos ejecutando el programa línea a Línea si hay alguna función definida por nosotros y queremos meternos dentro pulsamos F11 y para parar la ejecución en modo debug mayúsculas o shif t+ F11

Se para la ejecución en ese punto y nos muestra el valor de la variable temp que es el que hemos introducido.




Para seguir con la depuración podemos ir ejecutando F11 o los iconos del menú



F11->paso a paso

F10->paso por procedimiento

Shit+F11-> paso a paso para salir