Tema 1- Introducción a tecnología NET con Mono y fundamentos de C#

¿Qué es C# y NET?

C# es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

.NET es un framework de Microsoft que incluye programas, bibliotecas y lenguajes, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

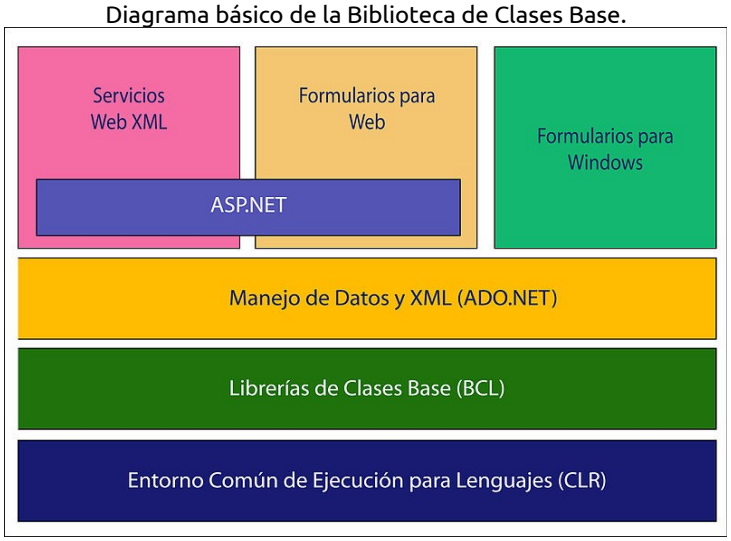
Los principales componentes de la plataforma .NET son:

- El conjunto de lenguajes de programación. **El que destaca es C#**.
- La biblioteca de clases base o BCL.
- **El entorno común de ejecución para lenguajes, o CLR** por sus siglas en inglés.

El **CLR** es el verdadero núcleo de .NET, entorno de ejecución en el que se cargan las aplicaciones desarrolladas. Permite integrar proyectos en distintos lenguajes como C++, Visual Basic, C#, entre otros. Esto lo hace compilando el código fuente de cualquiera de los lenguajes soportados en un código intermedio, el CIL (Common Intermediate Language). Para ejecutarse el CLR usa un compilador JIT (Just-In-Time) es el que genera el código máquina que se ejecuta en la plataforma del cliente.

La Biblioteca de Clases Base (BCL por sus siglas en inglés) maneja la mayoría de las operaciones básicas que se encuentran involucradas en el desarrollo de aplicaciones.

Diagrama básico de la Biblioteca de Clases Base.



Aunque C# forma parte de la plataforma .NET, C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Existe un compilador implementado que provee el marco **Mono**, el cual genera programas para distintas plataformas.

¿Qué es Mono?

Mono es el nombre de un proyecto de código abierto para crear herramientas libres, basadas en GNU/Linux y compatibles con .NET.

Mono posee importantes componentes útiles para desarrollar software:

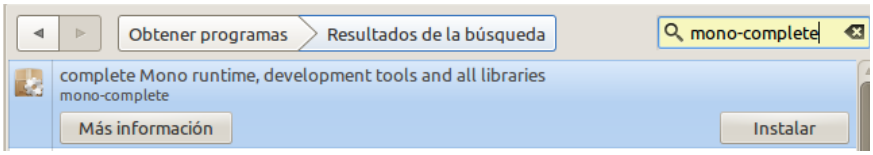
- Una máquina virtual de infraestructura de lenguaje común (CLI) que contiene un cargador de clases, un compilador en tiempo de ejecución (JIT), y unas rutinas de recolección de memoria.
- Una biblioteca de clases.
- Un compilador para el lenguaje C#.
- Es un proyecto independiente de la plataforma. Actualmente Mono funciona en GNU/Linux, OpenBSD, FreeBSD, UNIX, Mac OS X, Solaris y plataformas Windows.

Instalación del entorno Mono y el IDE MonoDevelop

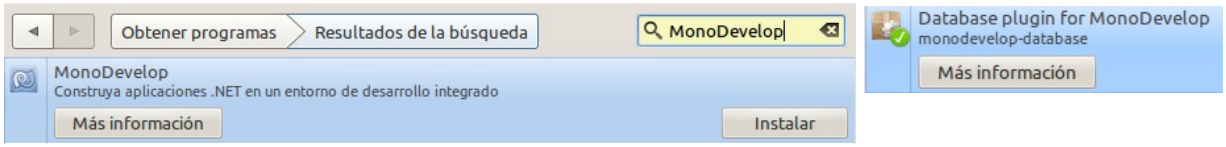
A partir de la versión 12.04 de Ubuntu Mono viene instalado por defecto. Si se usa una versión anterior hay que seguir las instrucciones de instalación.

Instalación en Ubuntu 10.04

Desde el centro de software buscar el paquete mono-complete. Esto instalará todos los paquetes dependencias que necesita el entorno Mono.



Instalar el IDE MonoDevelop también desde el centro de software. También se puede instalar el plugin o complemento para trabajar con Bases de datos.



Primera toma de contacto con el entorno de desarrollo MonoDevelop

Para empezar se va crear un programa de consola básico llamado “Hello World”, para ello:

- Abrir MonoDevelop
- Seleccionar desde menú archivo “Iniciar nueva solución”
- Elegir “Proyecto de consola” e indicar el título: “HolaMundo”
- Continuar, dejando en blanco las opciones de empaquetado y demás que propone.

Automáticamente se obtiene por defecto el código que imprime un saludo por consola.

En el archivo Main.cs se encuentra la clase principal.

```
using System;

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

Explicación

using System: Se usa el namespace System para hacer uso de Clases y métodos del sistema.

namespace HolaMundo: Se define un namespace propio.

La palabra clave **namespace** se utiliza para declarar un ámbito. Este ámbito permite organizar el código y proporciona una forma de crear tipos globalmente únicos y organizar sus múltiples clases.

class MainClass: Definición de clase.

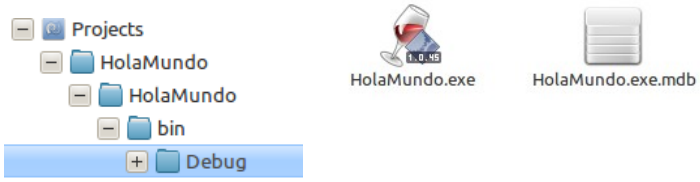
public static void Main(string[] args) Definición del método Main, este método puede recibir una serie de argumentos.

Console.WriteLine("¡Hola Mundo!"); En la clase Console hay un método WriteLine al que se le pasa una cadena y escribe en la consola dicha cadena, "¡Hola Mundo!". La clase Console pertenece al namespace System.

Compilar y ejecutar

Para compilar tan solo hay que usar F8 o desde menu Construir-> Construir Todo.
Para ejecutar el ejecutable con F5 o desde menú Ejecutar.

Al compilar se genera un ejecutable de tipo .exe que se puede encontrar en el directorio bin del proyecto.



El método Main no devuelve nada (de ahí la palabra clave void) pero puede recibir argumentos. Se pueden aceptar otras construcciones del método Main():

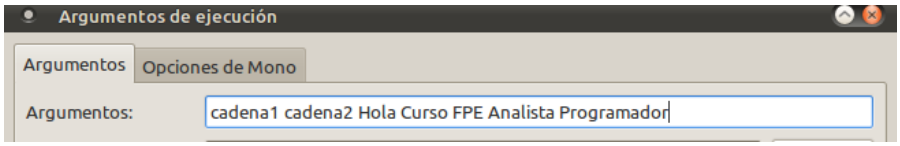
```
public static void Main ()
public static void Main (string[] args)
public static int Main ()
public static int Main (string[] args)


public static void Main (string[] args)
```

Esta variante toma un array(matriz) de cadenas, cada cadena del array se corresponde a un argumento de la línea de comandos suministrado al ejecutar el programa.
Para mostrar por la salida aparte del saludo las cadenas entregadas como argumentos habría que modificar el método Main:

```
class MainClass{
	public static void Main (string[] args){
		foreach (string cadena in args){
			Console.WriteLine(cadena);
		}
			Console.WriteLine ("Hello World!");
	}
}
```

Con la estructura iterativa **foreach** se recorre el array args y cada cadena(string) del array se imprime por la salida.

A la hora de ejecutar hay que seleccionar Ejecutar con parámetros personalizados e introducir los argumentos.



Todas las variantes del método Main() requieren la palabra clave static.

Espacios de nombres NAMESPACE

Así como los directorios permiten organizar archivos de modo que ficheros del mismo nombre convivan en directorios distinto, los espacios de nombres realizan una tarea similar, pero de manera lógica, para tipos de datos extendidos como clases y estructuras.

De este modo, dos clases distintas con el mismo nombre podrían ser utilizadas en el mismo proyecto sin que ocurra ningún error.
Los namespace se pueden anidar, por lo que puede haber unos dentro de otros.

Los espacios de nombres tienen las propiedades siguientes:

- Organizan proyectos de código de gran tamaño.
- El operador . delimita los espacios de nombres.

Namespace más populares de la librería BCL(clases base):

namespace	Descripción	Equivalente Java
System	Clase Base de la librería	java.lang
System.Collections	Clases e interfaces que definen colecciones de objetos como listas, colas y diccionarios.	java.util
System.Data	Acceso a datos	java.sql
System.Drawing	Acceso a funcionalidades gráficas de GDI+	java.awt
System.IO	Manejo de entrada y salida a archivos y flujos.	java.io
System.Net	Manejo de comunicación vía red pudiendo usar una gran variedad de protocolos.	java.net
System.Reflection	Acceso a información de clases cargada, se pueden crear e invocar tipos de modo dinámico.	java.lang.reflect
System.Text	Ofrece recursos para manipulación de texto en distintas codificaciones.	java.text
System.Threading	Recursos para programación de aplicaciones multihilo.	java.util.concurrent
System.Security	Provee acceso a funcionalidades de encriptación y configuración de seguridad del CLR.	java.security.javax.crypto
System.Windows.Forms	Manejo de ventanas y controles gráficos	java.swing
System.Xml	Manejo de datos en formato XML	java.xml

Con la sentencia using se ahorra el anteponer namespaces a cada una de las clases que se utilicen. Esta sentencia se usa en la cabecera del archivo:

```
using <namespace>;
```

El compilador buscará cada clase referenciada en el namespace global y en los especificados por la sentencia using.

Controlar el ámbito con namespace

La capacidad de crear ámbitos dentro del proyecto permite organizar el código y proporciona una manera de crear tipos únicos globales. En el ejemplo siguiente, una clase titulada SampleClass se define en dos namespace diferentes, uno anidado dentro de otro. El Operador punto se utiliza para diferenciar a qué método se llama.

```
using System;

namespace Padre
{
       class SampleClass{
        public void SampleMethod(){
            Console.WriteLine("SampleMethod dentro de Padre");
        }
    }

    namespace Hijo
    {
        // clase con el mismo nombre pero en namespace diferente
             class SampleClass{
            public void SampleMethod(){
                Console.WriteLine("SampleMethod dentro de Hijo");
            }
        }
    }

    class Program{
        static void Main(string[] args){
            // muestra "SampleMethod dentro de Padre"
            SampleClass outer = new SampleClass();
            outer.SampleMethod();

            // muestra "SampleMethod dentro de Padre"
            Padre.SampleClass outer2 = new Padre.SampleClass();
            outer2.SampleMethod();

            // muestra "SampleMethod dentro de Hijo"
            Hijo.SampleClass inner = new Hijo.SampleClass();
            inner.SampleMethod();
        }
    }
}
```

En el siguiente ejemplo, se muestran clases y espacios de nombres anidados. El nombre completo se indica como un comentario que sigue a cada entidad.

```
namespace N1       // N1
{
    class C1{      // N1.C1
        class C2{  // N1.C1.C2
        }
    }
    namespace N2  // N1.N2
    {
        class C2{ // N1.N2.C2
        }
    }
}
```

Se puede agregar un nuevo miembro de clase, C3, al espacio de nombres N1.N2 de la siguiente forma:

```
namespace N1.N2
{
    class C3 {  // N1.N2.C3
    }
}
```

Se puede crear un alias para un espacio de nombres. Por ejemplo, si utiliza un espacio de nombres escrito con anterioridad que contiene espacios de nombres anidados, puede declarar un alias para proporcionar una forma rápida de hacer referencia a uno en particular, de la siguiente manera:

```
using Product = Company.Dep.Pro;   // define alias que representa a un namespace
```

El operador :: se utiliza para hacer referencia a un alias de namespace y **global::** para hacer referencia al namespace global y el punto para calificar tipos o miembros.

No es correcto utilizar :: con un alias que hace referencia a un tipo en lugar de a un namespace. Por ejemplo:

```
using Alias = System.Console;

class TestClass{
    static void Main(){
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}
```

Se hace un alias de una clase de un namespace, por eso el alias hace referencia a un tipo y hay que usar el punto.

```
using Alias = System.Console;
```

Comentarios

En C# hay dos formas de escribir comentarios. La primera consiste en encerrar todo el texto que se desee comentar entre caracteres /* y */ siguiendo la siguiente sintaxis:

```
/* <comentario> */
```

Estos comentarios pueden abarcar tantas líneas como sea necesario. Para los comentarios de una sola línea sería de la siguiente manera:

```
// <comentario linea>
```

Documentación estructurada y generación en XML

Se pueden escribir comentarios estructurados en etiquetas y que éstos aparezcan como ayudas al escribir el nombre o parámetros de la clase, objeto, variable, etc. Además también se puede generar la propia documentación del programa basada exclusivamente en estos comentarios.

Ejemplo de documentación con tags descriptivos.

```
using System;

namespace ejemplo_02
{
	///<summary>
		///Descripcion corta de la clase CApp2
	///</summary>
	///<remarks>
		///añadir informacion extra
	///</remarks>
	///<example>
		///ejemplo de uso de la clase CApp2
		///<code>
		///codigo de ejemplo de uso
		///</code>
	///</example>
	class CApp2 {
		///<summary>
			///Descripcion corta de la propiedad
		///</summary>
		int var = 1;

		///<summary>
			///Descripcion corta del metodo
		///</summary>
		///<returns>
			///Describe lo que retorna el metodo
		///</returns>
		///<param name="archivo">
			///Descripcion del parametro archivo
		///</param>
		///<permission cref="LeeConfig()">
			///Tipo de permisos de acceso
		///</permission>
		public bool LeeConfig(string archivo) {
		bool c = false;
		return c;
		}
	}

	///<summary>
		///Descripcion corta de la clase Main
	///</summary>
	///<remarks>
		///añadir informacion extra
	///</remarks>
	class MainClass{
		public static void Main (string[] args){
			Console.WriteLine ("Hello World!");
		}
	}
}
```

Para generar la documentación en formato XML en el terminal hay que posicionarse en el directorio donde se encuentran los sources del proyecto y usar el siguiente comando para generar la documentación.

```
mcs -doc:Documentacion.xml Programa.cs
```

El XML generado para el ejemplo anterior tendría la siguiente estructura

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Main</name>
    </assembly>
    <members>
        <member name="T:ejemplo_02.CApp2">
            <summary>
            Descripcion corta de la clase CApp2
            </summary>
            <remarks>
            añadir informacion extra
            </remarks>
            <example>
            ejemplo de uso de la clase CApp2
            <code>
            codigo de ejemplo de uso
            </code></example>
        </member>
        <member name="F:ejemplo_02.CApp2.var">
            <summary>
            Descripcion corta de la propiedad
            </summary>
        </member>
        <member name="M:ejemplo_02.CApp2.LeeConfig(System.String)">
            <summary>
            Descripcion corta del metodo
            </summary>
            <returns>
            Describe lo que retorna el metodo
            </returns>
            <param name="archivo">
            Descripcion del parametro archivo
            </param>
            <permission cref="LeeConfig()">
            Tipo de permisos de acceso
            </permission>
        </member>
        <member name="T:ejemplo_02.MainClass">
            <summary>
            Descripcion corta de la clase Main
            </summary>
            <remarks>
            añadir informacion extra
            </remarks>
        </member>
    </members>
</doc>
```

Con una hoja de estilo se podría mostrar el documento XML en una web y proporcionar a los demás usuarios documentación actualizada del código.

El tag principal del XML es el elemento <members>. Este elemento contiene una etiqueta <member> para cada objeto documentado en el source. Esta etiqueta contiene un atributo llamado name que designa al elemento documentado. El valor de name empieza con un prefijo que describe el tipo de información:

Prefijo	Significado
E	Documentación de un evento.
F	Documentación de un campo(variable de clase)
M	Documentación de un método.
N	Documentación de un namespace.
P	Documentación de una propiedad.
T	Documentación de un tipo definido por el usuario(clase, interfaz, estructura, enumeración)
!	Error determinando prefijo correcto de ese miembro*.

*Las clases y estructuras tienen miembros que representan sus datos y comportamiento.

El valor del atributo name tiene esta estructura:

prefijo:namespace.clase.miembro

Trabajar con variables

Para declarar una variable se requiere especificar su tipo y un identificador:

tipo identificador;
```
int var;
```

Los identificadores deben comenzar por una letra, pueden contener guiones bajos, y no deben coincidir con una palabra reservada. El nombre de las variables es case-sensitive con lo cual hay diferencia entre minúsculas y mayúsculas pero no es aconsejable esta practica.

Para nombres compuestos de dos o mas palabras lo ideal es usar letras en mayúsculas al principio de cada palabra.

```
int unaVariableEntera;
```

Tipos de datos fundamentales

El tipo de dato es una manera de comunicarle al compilador cuánto espacio en memoria ocupará la variable y qué semántica tendrá el dato almacenado.

El compilador distingue entre los tipos de datos por valor y los tipos de datos por referencia. Los tipos de datos por valor serán almacenados en el stack(pila), mientras que los tipos por referencia serán almacenados en el heap(en el stack quedará una referencia de la ubicación en dicha zona de memoria).

Números enteros

Hay tres tipos de números enteros (short, int, long). La diferencia es la cantidad de memoria que reservan cada tipo.

short(2bytes) con un rango (-32767, 32767)
int(4bytes) con un rango mucho mas grande (-2147483647, 2147483647)

Los tipos enteros pueden tener signo o no, para declarar un entero sin signo se antepone una "u" al tipo. Un ejemplo:

ushort(2bytes) rango (0, 65535)

C# permite el uso de punteros pero su uso queda relegado a necesidades especificas, como interactuar con recursos no gestionados, además de tener que modificar parámetros de compilación.

El tipo de datos **byte** sirve para almacenar el valor de un byte, su rango es (0, 255).

Números no enteros

Hay tres tipos float, double y decimal. La diferencia es la precisión, esto tiene que ver con el "coma flotante" y el numero de decimales.

Tipo	Precisión
float	Precisión simple. Deben contener el agregado **f** en la parte decimal para que el compilador pueda tomarlas como float (**4.0f**)
double	Precisión doble (4.0)
decimal	Precisión máxima. Deben contener el agregado **M** en la parte decimal para que el compilador pueda tomarlas como decimal (**4.0M**)

Booleanos

Albergan dos valores: true o false.

```
bool a;
```

Caracteres

Para almacenar un carácter su tipo es:

```
char a;
```

Puede albergar caracteres de tipo Unicode.

Existen caracteres “especiales” que son interpretados por el compilador dentro de una cadena de texto, son conocidos como caracteres de escape:

Nombre	Escape
Línea nueva	\n
Tab horizontal	\t
Backspace	\b
Retorno de carro	\r
Barra invertida	\\
Comilla doble	\”

Para asignar un valor a una variable podemos hacerlo de varias formas:

```
int var = 1;

int var;

var = 1;
```

Constantes

Existen tres constantes:

Literales: Especificadas por medio de un valor escrito en el código.

```
int var = 1;

string nombre = “Paco”;
```

Simbólicas: son muy similares a las variables, con la diferencia de que a estas sólo se les podrá dar un valor en el momento de su declaración. Luego, cualquier intento de modificación será tomado como un error. Se debe anteponer la sentencia const antes del tipo.

```
const int unaConstante = 1;
```

Enumeradores: sirven para especificar un grupo de constantes entrechamente relacionadas.

```
enum Mes{
       Enero,
       Febrero,
       Marzo
}
```

Luego podríamos declarar una variable del tipo Mes:

```
Mes a;
a = Mes.Enero; // a solo podrá adoptar valores constantes en el enumerador
```

Conversiones de tipo de datos

C# es un lenguaje muy estricto en cuanto a conversiones de tipo. Existen dos tipos de conversiones:

Implícitas: son las que realiza el compilador sin requerir de ninguna sentencia adicional. Es una conversión en la nunca se pierde información ni precisión.

```
int var1 = Int32.MaxValue;
long var2 = var1;
```

Explicitas: cuando podría ocurrir pérdida de información o precisión el compilador exige utilizar la expresión de casting. Esto significa especificar el tipo de dato al cual deseamos llevar la expresión, ubicado entre paréntesis y antes de la expresión en cuestión:

```
long var3 = Int64.MaxValue;
int var4 = (int) var3;
```

En este caso no se produciría ningún error de compilación pero en la conversión de tipo habría una perdida de información importante. Si se diese salida a los valores obtendríamos:

9223372036854775807
-1

Para vigilar con estas perdidas se puede usar el bloque checker para lanzar la excepción *System.OverflowException: Number overflow* en el caso que haya perdidas en la conversión.

```
long var3 = Int64.MaxValue;
checked{
      int var4 = (int) var3;
}
```

También existe el bloque unchecked para que el código sea compilado aunque haya un error de conversión explicita.

La Clase Convert

Esta clase dentro del namespace System ofrece métodos estáticos para realizar conversiones:

```
bool var5 = 1; // error de compilador
bool var6 = Convert.ToBoolean(1); // Conversion a True
```

Operadores

Los operadores matemáticos son: + - * / %. Y se pueden combinar números y variables en las operaciones:

```
int var1 = 3 * 4; //almacena resultado en var1
int var2 = var1 / 3; //almacena resultado de var1 dividido entre 3
```

Con decimales:

```
float var3 = 3.4f / 2;//la f indica de tipo float
```

```
// combinacion de tipos de datos en operaciones
int var4 = 5;
float var5 = 4.3f;
float result = var4 * var5;
```

Cuando operamos el tipo de dato resultante estará en función de la operación realizada. En la división de dos números enteros dará como resultado un número entero aunque el tipo de dato del resultado sea de tipo float.

```
float var6 = 5 / 2; // el resultado obtenido sera 2
```

Para solucionar este problema hay que convertir al menos uno de los operandos en flotante.

```
float var7 = 5.0f / 2; // el resultado obtenido sera 2.5
```

Usar constantes en lugar de números literales es una buena practica para facilitar la comprensión del código.

```
const float pi = 3.1415f;
float radio = 2.5f;
float superficie = pi * (radio * radio);
```

CONTROL DE FLUJO DE EJECUCIÓN

Condicionales

Las expresiones a evaluar dentro de una sentencia **if** usualmente es una comparación utilizando los operadores:

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

En este ejemplo se obtiene el formato ASCII de la tecla de teclado, para ello se usa el método Peek para tomar un carácter desde la consola. Este carácter introducido se recoge en formato ASCII por eso var1 es de tipo int. En la sentencia condicional se comprueba si el código del carácter coincide con el carácter.

```
int var1 = Console.In.Peek();
if(var1 == 'x'){
	Console.WriteLine(var1);
}
```

Para probar el ejemplo después de compilar la solución desde la terminal ejecutar el .exe como se indica:

/Projects/ejemplos/ejemplos/bin/Debug$ mono ejemplos.exe

Podemos usar el **else** para ejecutar sentencias si la expresión es evaluada en falso:

```
int var1 = 120;
if(var1 == 'x'){
	Console.WriteLine("correcto");
}else{
	Console.WriteLine("no es correcto");
}
```

También se puede hacer uso de **else if** para diferentes casos de evaluación.

```
if(mes == 1){
	Console.WriteLine("Enero");
}else if(mes == 2){
	Console.WriteLine("Febrero");
}else if(mes == 3){
	Console.WriteLine("Marzo");
}else{
	Console.WriteLine("Nada");
}
```

Operadores lógicos

Se pueden crear expresiones más complejas combinando operadores lógicos:

Operadores	Descripción
&&	Y lógico (conjunción)
||	O lógico (disyunción)
!	Negación lógica

```
if(edad >= 18 && edad <= 40){
	Console.WriteLine("adulto");
}

if(mes == 2 || edad >= 18){
	Console.WriteLine("descuento");
}
```

Sentencia switch

En algunos casos es preferible reemplazar el uso de sentencias if anidadas por uso de la sentencia switch. Break se usa para salir de la sentencia switch y default es la opción que se toma si no hay ningún case que coincida con el valor.

```
switch(mes){
case 1:
	Console.WriteLine("Enero");
	break;
case 2:
	Console.WriteLine("Febrero");
	break;
default:
	Console.WriteLine("Ningun mes");
	break;
}
```

Para saltar de un case a otro se usa la sentencia **goto**

```
case 1:
	Console.WriteLine("Enero");
	goto case 2;
```

En el caso de que el tratamiento de una opción sea igual al de otra:

```
case 0:
case 1:
	Console.WriteLine("Enero");
	break;
```

No se pueden usar expresiones de evaluación en los case de sentencias switch

Sentencia bucle while

En este caso mientras la expresión asociada sea verdadera las sentencias se repetirán. Por ejemplo un programa que, mientras el usuario no ingrese una palabra determinada no termine:

```
string palabraClave = "";
while(palabraClave != "salir"){
	Console.WriteLine("Ingrese la palabra clave");
	palabraClave = Console.In.ReadLine();
}
Console.WriteLine("Correcto");
```

Sentencia do

La sentencia do es muy similar a while, solo que la evaluación de la expresión es al final y no al principio.

```
do{
	Console.WriteLine("Ingrese la palabra clave");
	palabraClave = Console.In.ReadLine();
}while(palabraClave != "salir");
Console.WriteLine("Correcto");
```

Sentencia for

for(<sentencia>; <expresion booleana>;<sentencia>)<sentencias a repetir>

La primera sentencia solo se ejecuta una vez. Luego, la expresión se evalúa antes de ingresar en cada bucle, y la segunda sentencia se ejecuta al finalizar cada bucle.

```
for(int i=0; i<5; i++){
	Console.WriteLine(i);
}
```

El valor de i se incrementa en una unidad después de cada ciclo.

El equivalente usando una sentencia while sería:

```
int a=0;
while(a<5){
	Console.WriteLine(a);
	a++;
}
```

La sentencia foreach

Esta sentencia es muy útil para recorrer colecciones y arrays.
Para recorrer un array por ejemplo sería:

```
int[] enteros = {500, 100, 200} ;
foreach(int num in enteros){
	Console.WriteLine("número {0}", num);
}
```

Para cada iteración la variable num tendrá el valor de un elemento del array.
Su equivalente con una sentencia for sería:

```
for(int i=0; i<enteros.Length; i++){
	Console.WriteLine("número {0}", enteros[i]);
}
```

Length sirve para saber la cantidad de elementos que posee el array.

Las sentencias break y continue

break se utiliza para salir del bucle en el cual nos encontramos, independientemente de la expresión que se evalúa.

```
for(int i=0; i<10; i++){
	Console.WriteLine("número {0}", i);
	if(i>5)
		break;
}
```

La sentencia **continue** permite saltar lo que resta de una iteración dentro el bucle para seguir con la próxima(sin salir del bucle).

```
for(int i=0; i<10; i++){
	Console.WriteLine("punto a");
	if(i>3)
		continue;
		Console.WriteLine("punto b");
}
```

ACTIVIDADES:

1.- ¿Donde se encuentran los errores?. Indicarlo.

a)

```
int n1 = 10;
double n2 = n1;
float n3 = n2;
```

b)

```
int valor;
if(valor == 0)
      Console.WriteLine("el valor es cero");
```

c)

```
for(int i=0; i<5, i++)
      Console.WriteLine("--");
```

d)

```
int num = 0;
if(num == 0)
      Console.WriteLine("Es valor");
      Console.WriteLine("es cero!");
else
      Console.WriteLine("valor distinto de cero");
```

2.-¿Para qué se utilizan los namespace?

3.-¿Cómo se puede convertir un string que posea un número a una variable numérica?