C#

Índice

1) El primer programa

```
class HellowWorld
{
      public static void Main ()
      {
            System.Console.WriteLine("Hello World!");
      }
}
```

Este es un programa de una sola sentencia, que sirve para mostrar en la consola la cadena "Hello World!" con la función WriteLine. Si quisieramos mostrar el valor de una variable con esta función se haría:

```
System.Console.WriteLine("Ejemplo: {0} {1} {2} {0}", var0, var1, var2);
```

La función Main

```
public static {void | int} Main {() | (string [] Arguments)}
{
   // Código del programa...
}
```

Comentarios

```
// Una sola línea de código...
/* Varias líneas
   de código... */
```

Generación de documentación en XML

Para ello tiene que realizar dos cosas:
- Poner los comentarios con ///
- E indicar la opción /doc al compilador:
  csc /doc:fich.xml fich.cs

(Páginas 65-73)

2) Trabajar con variables

```
tipo identificador;
```

Reglas para los identificadores

- Ha de empezar con una letra o un guión bajo.
- Tras esto se pueden usar letras, números o el guión bajo.
- Ejemplos: pos_x, pos_y, _nombre, _sueldo_20, Apellido1, ColorCirculo.

Tipos de variable

+ Enteros sin signo: `byte` (8b), `ushort` (16b), `uint` (32b), `ulong` (64b).
+ Enteros con signo: sbyte (8b), `short` (16b), `int` (32b), `long` (64b).
+ Reales: `float` (32b), `double` (64b), `decimal` (128b).
+ Otros: `char` (16b, un carácter Unicode: 'a', 'あ', 'Ñ'), `bool` (`true` o `false`), `void`.

Declaración

```
tipo identificador [= valor];
```

Si no se indica un valor a la variable, el compilador da valores por defecto a las variables declaradas al nivel de la clase. Son `0`, `0.0`, `false`.

Las matrices

Declaración:

```
tipo [] identificador [= {valor, ...}];
tipo [,] identificador [= {{...}, {...}, ...}];
tipo [[,[...]]] identificador [= {...}];
```

Reserva de espacio:

```
identificador = new tipo{[valor] | [valor, ...]};
identificador[valor, ...] = new tipo[valor];
```

Acceso al contenido:

```
identificador[valor];
identificador[valor, ...];
```

Conversión de tipos

```
variable = (tipo) variable2; // System.Convert.To<tipo>(...);
```

Cadenas

```
string var [= "cadena"];
```

+ Caracteres especiales

\t	tabulación
\r	retorno de carro
\v	tabulación vertical
\f	caracter de impresión de página
\n	salto de linea
\x	caracter ASCII en hexadecimal (\xFF)
\u	caracter Unicode (\uFFFF)
\\	caracter barra '\'

+ Desactivación de los caracteres especiales en cadenas

```
cadena = "C:\\MiDir\\mifich.txt";
cadena = @"C:\MiDir\mifich.txt";
```

+ Acceso a un caracter

```
char car = cadena[0];
```

Enumeraciones (System.Enum)

```
public enum Nombre
{
   identificador [= valor],
   ...
}
```

3) Expresiones

Literales

Cadenas	"hola"
Enteros	20
sin signo	20U
Booleanos	true (= -1), false (= 0)
Entero largo	20L
sin signo	20LU, 20UL
Reales	1.0 7.5e+2 7.5e-2
float	1.0F
double	1.0D
decimal	1.0M
Caracter	'a'
Hexadecimal	0xFF
Nulo	null

Operadores especiales

`.`	Acceso a los miembros de un tipo: `obj.función(); obj.variable;`
`[]`	Indexador de matrices: `var[valor];`
`this`	Referencia a la propia clase: `this.miembro;`
`base`	Referencia a la clase padre: `base.miembro;`
`new`	Petición de memoria.

`typeof(tipo)`	Devuelve un objeto de tipo System.Type, con información del tipo.
`checked(expresión)`	Lanza una excepción en caso de fallo en la operación.
`unchecked(expresión)`	Por defecto.

Operadores

+ Unarios:	`+ - ! ~ ++ --`
+ Aritméticos:	`* / % + - << >>`
+ Asignación:	`= *= /= %= += -= <<= >>= &= |= ^=`
+ Relacionales:	`== != > >= < <=`
+ Lógicos:	`& ^ | && ||`
+ Condicional:	`condición ? expresiónTrue : expresiónFalse;`

4) Control de flujo

Nota: Todas las instrucciones terminan en ;

Constantes

```
const tipo nombre = valor;
```

Condicionales

```
if (condición) {
    ...;
} [else {
    ...;
}]
```

```
switch (variable) { //entero, caracter o cadena
    case valor:
        ...;
        [break;]
    [...]
    [default:
        ...;]
}
```

Repetitivas

```
while (condición) { ... }
do { ... } while (condición);
for (iniciador; condición; paso) { ... }
foreach (variable in vararray) { ... }
```

Salto

```
break;
continue;
goto etiqueta; //etiqueta: instrucción;
```

Otras

```
checked { ... }
using librería;
namespace nombre { ... }
```

5) Metodos

```
tipo nombre (tipo nombre, ...) {
    ...;
    [return valor;]
}
```

Casos especiales

+ De entrada: `tipo nombre`
+ De salida: `out tipo nombre`
+ De referencia: `ref tipo nombre`
+ De parametros: `params tipo[] nombre`

C# soporta la sobrecarga de métodos.

6) Estructuras

```
struct nombre [: interfaz, ...] {
    public declaraciónElemento;
    ...;
}
```

Uso: `variable.campo;`

Los elementos pueden ser variables y métodos, estos pueden actuar de constructores, poniéndole al método el nombre de la estructura, para luego crear la variable así:

```
nombre variable [= new nombre(...)];
```

Indizadores

```
struct ArrayEnteros {
   public int [] dato;
   public int this [int indice] {
        get { return dato[indice]; }
        set { dato[indice] = value; }
   }
}

// Código en el main:
ArrayEnteros aux = new ArrayEnteros();
aux.dato = new int[2];
aux[0] = 1;
aux[1] = 2;
```

Uso de los tipos simples como estructuras

```
sbyte   → System.SByte      byte    → System.Byte
short   → System.Int16      ushort  → System.UInt16
int     → System.Int32      uint    → System.UInt32
long    → System.Int64      ulong   → System.UInt64
char    → System.Char       bool    → System.Bool
float   → System.Single     double  → System.Double
decimal → System.Decimal
```

7) Clases y objetos

Cosas que se permiten: Abstracción, tipos de datos abstractos, encapsulación, herencia, polimorfismo.

Declaración

```
[public | private] class NombreClase {
   [public | private | protected] declaraciónMiembro;
   ...;
}
```

Un miembro puede ser una variable, una función o estructura. También constantes, campos, métodos, propiedades, eventos, indizadores, operadores, constructores, destructores y tipos.

Constantes

```
[public | private | protected] [static] const tipo nombre = valor;
```

Campos

```
[public | private | protected] [static] [readonly] tipo nombre;
```

`readonly` = Lo convierte en un campo de solo lectura. Solo se le puede dar valor en el constructor de la clase.

Métodos

```
[public | private | protected] [static] tipo nombre (...) { ... }
```

Propiedades

```
[public | private | protected] tipo nombre {
    [get { return this.variable; }]
    [set { this.variable = value; }]
}
```

Indizadores

```
[public | private | protected] tipo this [int indice] {
    [get { return this.variable[indice]; }]
    [set { this.variable[indice] = value; }]
}
```

Constructores

```
[public | private | protected] NombreClase (...) { ... }
```

Destructores

```
~NombreClase () { ... }
```

this

`this` es una referencia a la propia clase.

static

Después del modificador de visibilidad (`public`, `private`, `protected`) se puede poner en campos, constantes, métodos. Esto sirve para que solo haya una instancia en memoria de dichos elementos. Para referenciarlos se usa:

```
NombreClase.miembro;
```

Que no es lo mismo que el nombre de la variable objeto instanciada, sino el nombre de la clase.

Instanciar una clase

```
NombreClase variable = new NombreClase(...);
```

8) Sobrecarga de operadores

Operadores unarios

```
public static tipo operator <operador> (tipo opde)
{
    ...;
    return valor;
}
```

Operadores binarios

```
public static tipo operator <operador> (tipo opiz, tipo opde)
{
```

```
    ...;
    return valor;
}
```

Operadores que no se pueden sobrecargar

+ Asignación: `=`
+ Condicional: `&&` `||` `?:`
+ Otros: `new` `typeof` `sizeof` `is`

9) Herencia

Como heredar

```
[public | private] class NombreClase : NombrePadre {
    ...;
}
```

Modificadores ambito

Visibilidad	Fuera	Hijos	Nietos
`public`	Sí	Sí	Sí
`private`	No	No	No
`protected`	No	Sí	Sí

Existe un cuarto modificador de ámbito denominado `internal`, que hace visible al miembro solo dentro del mismo fichero binario.

Métodos virtuales y de reemplazo

```
ámbito vitual tipo función (...) { ... }   // Clase padre
ámbito override tipo función (...) { ... } // Clase hija
```

Las clases hijas tienen que sobreescribir los métodos virtuales de la clase padre.

Clases y métodos abstractos

```
ámbito abstract class nombre { ... }
ámbito abstract tipo función (...);
```

Al no tener definición, los hijos deben definirlas.

Propiedades e indizadores heredados

Se pueden usar las palabras `virtual`, `override` y `abstract`, con propiedades e indizadores. Funcionando del mismo modo que con los métodos.

base

La palabra clave `base` es una referencia a la clase padre. En el constructor se puede hacer uso de `base` para llamar al constructor padre:

```
ambito NombreClase (...) : base (...) { ... }
```

También se pueden llamar a elementos:

```
base.miembro;
```

Clase sellada

Prohibe que se herede de dicha clase:

ámbito sealed class nombre { ... }

La clase object

Si se declara: class nombre { ... }, esto sería igual que poner:

```
class nombre : System.Object { ... }
class nombre : object { ... }
```

La clase Object tiene los siguientes elementos:

```
public virtual bool Equals (object obj);
public virtual int GetHashCode ();
public Type GetType ();
public virtual string ToString ();
protected virtual void Finalize ();
protected Object MemberWiseClone ();
```

10) Espacios de nombre

```
namespace nombre { ... }
```

Puede contener clases, estructuras, interfaces, enumeraciones y delegados. Para hacer referencia a un elemento se usa: nombre.elemento;

using

```
using nombreDeEspacio[.elemento];
```

Añadiendo ese tipo de sentencia al principio del programa, se incluye dicho elemento en nuestro programa, pero la sentencia using también sirve para:

```
using nombreAlias = nombreDeEspacio.elemento;
```

Con eso se crea un alias, que se puede utilizar para referenciar todo tipo de elementos dentro de un namespace (ello incluye los propios espacios de nombre que se incluyan dentro). Esto a fin de cuentas acaba formando lo que se conoce como una biblioteca.

Bibliotecas de .NET

System: Funcionalidades básicas.
- CodeDOM: Clases que representan los elementos de un documento de código fuente.
- Collections: Listas, colas, matrices, tablas, hash y diccionarios.
- ComponentModel: Clases que se usan para crear componentes y controles durante el tiempo de diseño y ejecución.
- Data: Aquitectura ADO.NET.
- Diagnostics: Clases para detectar errores en las aplicaciones.
- Drawing: Clases para el uso del GDI.
- IO: Clases para el flujo de datos.
- Messaging: Colas de mensajes.
- Net: Clases para operaciones en red.
- Reflection: Vistas de tipos.
- Resources: Gestión de recursos especiales.
- Runtime: Tiene diversas bibliotecas interesantes.
- Security: Acceso a la seguridad.

- `Text`: Trabajo con el texto en diversas codificaciones como ASCII, Unicode o UTF-8.
- `Threading`: Multihilo.
- `Timers`: Para realizar eventos en un tiempo determinado.
- `Web`: Protocolo http implementado.
- `Windows.Forms`: Clases para crear aplicaciones windows.
- `Xml`: Para el manejo de ficheros xml.

11) Interfaces

```
interface inombre [: ipadre, ...] {
      tipo método (...);
      tipo propiedad { get; set; }
      tipo this [int index] { get; set; }
      event EentHandler ClickEvent;
      ...;
      // Nueva declaración: Para redefinir de nuevo algo definido ya
      //                    en los padres, como en la herencia de clases.
}
```

Implementación

```
class cnombre [: [cpadre,] [inombre, ...]] { ... }
```

Dentro de la clase se puede referenciar los elementos de una interfaz como: `inombre.elemento`.

is

(`var is inombre`) = `true` o `false`

Indica si una variable objeto contiene una interfaz.

as

```
inombre nombre;
nombre = obj as inombre;
if(nombre != null) nombre.elemento;
```

Ámbito

```
[public | private | protected | internal] interface inombre [:...] {
    ...;
}
```

Interfaces . NET (Páginas 317-325)

12) Eventos y delegados

Definir delegados y eventos

```
ámbito delegate tipo nombre (...);

ambito class cnombre {
    ...;
    ámbito event nombre enombre;
}
```

Instalar eventos

+ Creamos una funcón con la misma forma que el delegado:
  `ambito tipo fnombre (...) {...}`
+ Luego creamos una asociación:
  `nombre manejador;`
  `manejador = new nombre(fnombre);`
+ Y para añadir un evento:
  `enombre += manejador;`
+ Para borrarlo:
  `enombre -= manejador;`
+ Para generar un evento:
  `enombre(...);`

Ejemplo estandar

```
using System;

public delegate void ImparHandler (object origen,
                                   NumberEventArgs args);

class Contador {
   public event ImparHandler OnImpar;

   public Contador () {
        this.OnImpar = null;
   }

   public void Contar100 {
        for(int n = 0; n <= 100; n++) {
             if(n % 2 == 0) {
                  if(this.OnImpar != null) {
                       NumberEventArgs args = new NumberEventArgs(n);
                       this.OnImpar(this, args);
                  }
             }
        }
   }
}

public class NumberEventArgs : EventArgs {
   private int num;

   public NumberEventArgs (int numero) { this.num = numero; }

   public int Number {
        get { return this.num; }
   }
}
```

```
class CManejados {
   public void Manejador (object obj, NumberEventArg args) {
        Console.WriteLine(args.Number);
   }
}

class Principal {
   public static void Main () {
        Contador i = new Contador ();
        CManejador handler = new CManejador();
        i.OnImpar += new ImparHandler(handler.Manejador);
        i.Contar100();
   }
}
```

Descriptores de acceso

```
public event ImparHandler OnImpar {
   add { AddToList(value); }
   remove { RemoveFromList(value); }
}
```

(Nota: Value es de tipo Delegate.)

Modificadores

Se pueden usar `static`, `virtual`, `override` y `abstract` con eventos.

13) Excepciones

```
try {
     // Código "peligroso".
} [catch (Exception nombre) {
     // Código para manejar el error.
}] ... [catch {
     // Código por defecto para manejar errores.
}] [finally {
     // Bloque para liberar recursos, como cerrar archivos.
     // En realidad es un fragmento de código que siempre se ejecutará
     // después de procesar los errores, aunque se haga un throw.
}]
```

La clase exception (`System.Exception`)

+ `HelpLink`: Vínculo de ayuda.
+ `Message`: Mensaje de error.
+ `Source`: Nombre del objeto que lanzó el error.
+ `StackTrace`: Lista de las llamadas al método en la pila.
+ `TargetSite`: El nombre del método que inició la excepción.

Excepciones de .NET

+ `OuOfMemoryException`: Memoria agotada.

+ `StackOverflowException`: Desbordamiento de la pila.
+ `NullReferenceException`: Puntero nulo.
+ `TypeInitializationException`: Error en la inicialización.
+ `InvalidCastException`: Error en el cast.
+ `ArrayTypeMismatchException`: Error de tipos en un array.
+ `IndexOutOfRangeException`: Acceso fuera del array (violación de la memoria).
+ `DivideByZeroException`: División entre cero.
+ `OverflowException`: Desbordamiento.

Excepciones propias

```
class nombre : ApplicationException {
   public nombre (...) : base (...) { ... }
   ...;
}

// base puede recibir como argumentos:
// (string message)
// (string message, Exception innerException)
```

Para lanzar una excepción se emplea:
`throw new nombre (...);`

14) Trabajar con atributos

(Páginas 379-397)
(`System.Runtime.InteropServices` → `DllImportAttribute`)

15) Código no seguro

+ Comando de compilación: `csc /unsafe`
+ Declaración: `tipo * nombre;`
+ Asignación:

```
nombre = &var; // & → da al direción de memoria.
var = *nombre; // * → lo apuntado por nombre.
nombre = new tipo;
nombre = new tipo[];
nombre = new tipo();
```

+ Acceso a los elementos: `nombre->elemento;`
+ Acceso al contenido:

```
nombre[pos];
*(nombre + pos);
```

+ `sizeof(tipo)` → Nº de Bytes del tipo.
+ `stackalloc tipo [val]` → Pide una región de memoria.

16) Excentricidades

+ Errores de cast con objetos (424-425).
+ Las estructuras con atributos necesitan un new, como los objetos.

+ Como pasar clases derivadas (429-431).

17) Windows Forms

+ Namespace: `System.Windows.Form`
+ Opciones de compilado: `[STAThread]`
+ `Form`: Clase con la lógica del formulario.
  `.Text = "Título de la ventana";`
+ `Application`: Clase con la lógica para crear una aplicación.
  `.Run(new miForm()); // Ejecuta un formulario.`

Añadir información de la versión

```
[assembly: info("cadena")]
AssemblyTitle("Título")
AssemblyDescription("Descripción")
AssemblyConfiguration("retail")
AssemblyCompany("Compañía")
AssemblyProduct("Nombre del producto")
AssemblyCopyright("(c) ...")
AssemblyTrademark("Marca registrada")
AssemblyCulture("es-es")
AssemblyVersion("1.0.0.0")
```

Eventos (Application)

+ ApplicationExit (EventHandler)
+ Idle (EventHandler)
+ ThreadException (ThreadExceptionEventHandler)
+ ThreadExit (EventHandler)

Propiedades (Application)

+ AllowQuit
+ CommonAppDataRegistry
+ CommonAppDataPath
+ CompanyName
+ CurrentCulture
+ CurrentInputLanguage
+ ExecutablePath
+ LocalUserAppDataPath
+ MessageLoop
+ ProductName
+ ProductVersion
+ SafeTopLevelCaptionFormat
+ StartupPath
+ UserAppDataPath
+ UserAppDataRegistry

Métodos

+ AddMessageFilter (objeto) → Añade un filtro para controlar los mensajes.

```
class A : IMessageFilter { ...
        public bool PreFilterMessage (ref Message m) { ... }
}
```

+ DoEvents () → Procesa los mensajes de la cola.
+ Exit () → Cierra la aplicación.
+ ExitThread () → Cierra la aplicación y sus subprocesos.
+ OleRequired () → Inizializa OLE.
+ OnThreadException () → Lanza un evento ThreadException.
+ RemoveMessageFilter (objeto)
+ Run (objForm)

18) Archivos

Acceso binario (FileStream (path, FileMode.elemento, FileAccess.elemento))

+ BinaryWriter (FileStream obj)
  - BaseStream: Permite el acceso a la secuencia subyacente.
  - close ()
  - Flus ()
  - Seek ()
  - Write (object)

+ BinaryReader (FileStream obj)
  - Close ()
  - PeekChar (): Lee un char sin avanzar el puntero.
  - ReadChar ()
  - ReadBoolean ()
  - ReadByte ()
  - ReadInt16 ()
  - ReadInt32 ()
  - ReadInt64 ()

Para supervisar los cambios → FileSystemWatcher

Manipular ficheros

FileInfo (string path), FileSystemInfo, DirectoryInfo.

- CopyTo (string nuevoPath[, bool sobreescribir])
- Delete ()
- Attributes → FileAttributes
- MoveTo (string path[, bool sobreescribir])

Secuencias (Stream)

Stream → FileStream, MemoryStream, NetworkStream, CryptoStream, BufferedStream.

- Read (...);
- Write (...);

Lectores TextReader → StreamReader, StringReader, ...

Escritores TextWriter → IndentedTextWriter, StreamTextWriter,
StringTextWriter, HttpTextWriter,
HtmlTextTextWriter, ...

Xml → XmlWriter, XmlReader (System.Xml).
+ WriteStartDocument ();
+ WriteComment ("Comentario");
+ WriteStartElement ("Etiqueta");
+ WriteElementString ("Etiqueta", "Valor");
+ WriteEndElement ();
+ WriteEndDocument ();

19) GDI+

Librería: System.Drawing
Método: OnPaint
Clases: Graphics, Bitmap, Image, Brush, Font, ...

Graphics:
+ FillRectangle (...);
+ DrawString (...);
+ DrawImage (...);

Bitmap:
+ GetPixel (...);
+ SetPixel (...);

20) Trabajar con ensamblados

Assembly = Nos permite manipular y crear ensamblados en tiempo de ejecución.

Reflexión = Con esto se puede crear código en tiempo de ejecución.
(System.Reflection.Emit) (Páginas 700-702)

21) Multiproceso

Thread
Miembros: CurrentContext, CurrentThread, ResetAbort, Sleep,
ApartmentState, IsAlive, IsBackground, Name, Priority, ThreadState,
Abort, Interrupt, Join, Resume, Start, Suspend.

Sincronismo
Clases: Monitor, ...
lock (this) { ... } → Bloqueo con exclusión mutua.

22) Trabajar con COM

(Páginas 727-744)