Anexo V

C#.NET

Guion-resumen

1. **Introducción a C#.NET**
2. **Escritura y compilación de una aplicación básica**
3. **Espacios de nombres**
4. **Aplicaciones con argumentos**
5. **Tipos básicos**
6. **Instrucciones condicionales**
7. **Instrucciones iterativas**
8. **Ejemplos**

1. Introducción a C#.NET

C# (pronunciado en inglés "C Sharp" y en español "C Almohadilla") es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET. Como ya se ha visto, es posible escribir código para la plataforma .NET en muchos otros lenguajes, como Visual Basic.NET o JScript.Net. C# es el único que ha sido diseñado específicamente para ser utilizado en esta plataforma. Microsoft suele referirse a C# como el **lenguaje nativo de .NET**, gran parte de la librería de clases base de .NET ha sido escrito en este lenguaje. Nació en el año 2001 de la mano de Microsoft y se trata de un lenguaje influido por C++ (el código nos recuerda mucho su uso), JAVA, Delphi y Eiffel. Aunque C# forma parte de la plataforma.NET, esta es una interfaz de programación de aplicaciones, mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Es posible implementar compiladores que no generen programas para dicha plataforma, sino para una plataforma diferente como Linux.

En la web de Microsoft podemos leer de C#: "es un lenguaje orientado a objetos sencillo, moderno, amigable, intuitivo y fácilmente legible que ha sido diseñado con el objetivo de recoger las mejores características de muchos otros lenguajes, fundamentalmente Visual Basic, JAVA y C++, y combinarlas en uno solo en el que se unan la alta productividad y facilidad de aprendizaje de Visual Basic con la potencia de C++".

Como curiosidad: El símbolo # viene de sobreponer "++" sobre "++" y eliminar las separaciones, indicando así su descendencia de C++.

Los programas de C# se ejecutan en .NET Framework, que incluye un sistema de ejecución virtual denominado Common Language Runtime (CLR) y un conjunto unificado de bibliotecas de clases. CLR es la implementación comercial de Microsoft de Common Language Infrastructure (CLI), norma internacional que constituye la base para crear entornos de ejecución y desarrollo en los que los lenguajes y las bibliotecas trabajan juntos sin problemas.

El código fuente escrito en C# se compila en un lenguaje intermedio (IL) conforme con la especificación CLI. El código de lenguaje intermedio, junto con recursos, tales como mapas de "bits" y cadenas, se almacena en disco en un archivo ejecutable denominado ensamblado, cuya extensión es .exe o .dll generalmente. Un ensamblado contiene un manifiesto que ofrece información sobre los tipos, la versión, la referencia cultural y los requisitos de seguridad del ensamblado.

Cuando se ejecuta un programa de C#, el ensamblado se carga en CLR, con lo que se pueden realizar diversas acciones en función de la información del manifiesto. A continuación, si se cumplen los requisitos de seguridad, CLR realiza una compilación Just In Time (JIT) para convertir el código de lenguaje intermedio en instrucciones máquina nativas. CLR también proporciona otros servicios relacionados con la recolección automática de elementos no utilizados, el control de excepciones y la administración de recursos. El código ejecutado por CLR se denomina algunas veces "código administrado", en contraposición al "código no administrado" que se compi-

la en lenguaje máquina nativo destinado a un sistema específico. En el diagrama siguiente se muestran las relaciones en tiempo de compilación y tiempo de ejecución de los archivos de código fuente de C#, las bibliotecas de clases base, los ensamblados y CLR.

La interoperabilidad del lenguaje es una función clave de .NET Framework. Como el código de lenguaje intermedio generado por el compilador de C# cumple la especificación de tipos común (CTS), este código generado en C# puede interactuar con el código generado en las versiones .NET de Visual Basic, Visual C++, Visual J# o cualesquiera de los más de 20 lenguajes conformes con CTS. Un único ensamblado puede contener varios módulos escritos en diferentes lenguajes .NET, y los tipos admiten referencias entre sí como si estuvieran escritos en el mismo lenguaje.

Principales características de C#:

— C# es un lenguaje **simple, moderno, de propósito-general** de programación orientada a objetos.

— Dispone de todas las características propias de cualquier lenguaje orientado a objetos: **encapsulación, herencia y polimorfismo**.

— Ofrece un **date de programación orientada a objetos homogéneo**, en el que todo el código se escribe dentro de clases y todos los tipos de datos, incluso los básicos, son clases que heredan de **System.Object** (por lo que los métodos definidos en esta son comunes a todos los tipos del lenguaje).

— Permite definir **estructuras**, que son clases un tanto especiales: sus objetos se almacenan en pila, por lo que se trabaja con ellos directamente y no referencias al montículo, lo que permite accederlos más rápido. Sin embargo, esta mayor eficiencia en sus accesos tiene también sus inconvenientes, fundamentalmente que el tiempo necesario para pasarlas como parámetros a métodos es mayor (hay que copiar su valor completo y no solo una referencia) y no admiten herencia (aunque sí implementación de interfaces).

— Es un **lenguaje fuertemente tipado**, lo que significa que controla que todas las conversiones entre tipos se realicen de forma compatible, lo que asegura que nunca se acceda fuera del espacio de memoria ocupado por un objeto. Así se evitan frecuentes errores de programación y se consigue que los programas no puedan poner en peligro la integridad de otras aplicaciones.

— Tiene a su disposición un **recolector de basura** que libera al programador de la tarea de tener que eliminar las referencias a objetos que dejen de ser útiles, encargándose de ello este y evitándose así que se agote la memoria porque al programador olvide liberar objetos inútiles o que se produzcan errores porque el programador libere áreas de memoria ya liberadas y reasignadas.

— Incluye soporte nativo para **eventos y delegados**. Los delegados son similares a los punteros a funciones de otros lenguajes como C++, aunque más cercanos a la orientación a objetos, y los eventos son mecanismos mediante los cuales los objetos pueden notificar de la

ocurrencia de sucesos. Los eventos suelen usarse en combinación con los delegados para el diseño de interfaces gráficas de usuario, con lo que se proporciona al programador un mecanismo cómodo para escribir códigos de respuesta a los diferentes eventos que puedan surgir a lo largo de la ejecución de la aplicación (pulsación de un botón, modificación de un texto, etc.).

— Incorpora **propiedades**, que son un mecanismo que permite el acceso controlado a miembros de una clase tal y como si de campos públicos se tratasen. Gracias a ellas se evita la pérdida de legibilidad que en otros lenguajes causa la utilización de métodos *Set() y Get()* pero se mantienen todas las ventajas de un acceso controlado por estos.

— Permite la **definición del significado de los operadores básicos** del lenguaje (+, -, *, &, ==, etc.) para nuestros propios tipos de datos, lo que facilita enormemente tanto la legibilidad de las aplicaciones como el esfuerzo necesario para escribirlas. Es más, se puede incluso definir el significado del operador ***[]*** en cualquier clase, lo que permite acceder a sus objetos tal y como si fuesen tablas. A la definición de este último operador se le denomina **indizador**, y es especialmente útil a la hora de escribir o trabajar con colecciones de objetos.

— Admite unos elementos llamados **atributos,** que no son miembros de las clases sino información sobre estas que podemos incluir en su declaración. Por ejemplo, indican si un miembro de una clase ha de aparecer en la ventana de propiedades de Visual Studio.NET, cuáles son los valores admitidos para cada miembro en esta, etc.

— El lenguaje provee soporte para principios de **ingeniería de software** tales como revisión estricta de los tipos de datos, revisión de límites de arrays, detección de intentos de usar variables no inicializadas y recolección de basura automática.

— Es usado para desarrollar **componentes de software** que se puedan usar en ambientes distribuidos. C# es adecuado para escribir aplicaciones desde las más grandes y sofisticadas como sistemas operativos hasta las más pequeñas funciones.

— Soporte para **internacionalización**.

2. Escritura y compilación de una aplicación básica

Este programa lo único que hace al ejecutarse es mostrar por pantalla el mensaje "Hola Mundo":

```
// Lolo1.cs
    class Lolo1
    {
```

```
    public static void Main()
        {
            System.Console.WriteLine("¡Hola Mundo!");
        }
}
```

Si almacenamos este código en un fichero de texto plano de nombre **Lolo.cs** podemos compilarlo abriendo una ventana de consola (MS-DOS) y, tras colocarnos en el directorio donde hayamos almacenado el fichero, ejecutarlo: csc Lolo1.cs.

Csc es el nombre del compilador de C#, y la orden anterior simplemente indica que deseamos compilar el fichero de código fuente **Lolo.cs**, y tras ejecutarla el compilador generará un fichero de nombre **Lolo.exe** que contendrá el ejecutable de nuestra sencilla aplicación de ejemplo.

Para ejecutarlo basta escribir: *Lolo1*

El resultado que veremos por la pantalla será: ¡Hola Mundo!

Una vez que ya sabemos cómo compilar y ejecutar aplicaciones escritas en C#, es el momento de analizar detenidamente el significado del código anterior:

```
1:  class Lolo1
2:  {
3:      public static void Main()
4:      {
5:          System.Console.WriteLine("¡Hola Mundo!");
6:      }
7:  }
```

Nótese que se ha eliminado la línea *// Lolo.cs*: Ejemplo típico que aparecía en la versión original del código. Esto se debe a que es solo un comentario que se introduce para facilitar la legibilidad del código a los lectores pero que el compilador ignora completamente, por lo que no la incluiremos en el presente análisis. Cabe señalar que en C# se considera comentario a todo aquel texto comprendido entre los caracteres // y el final de la línea donde aparecen y a todo texto comprendido entre los caracteres /* y */.

Como se comentó en la introducción, todo el código escrito en C# se ha de escribir dentro de una clase. Así, en la línea **1**: se dice que lo que a conti-

nuación se incluirá es la definición de una clase (**class**) a la que le daremos el nombre de *Lolo1*, estando la definición de la misma comprendida entre la primera llave que aparezca (línea **2**:) y su correspondiente llave de cierre (línea 7:).

Lo que dentro de la definición de la clase se dice (línea **3**:) es que va a tener un método de nombre *Main* cuyo código es el indicado entre la próxima llave de apertura (línea **4**:) y su respectiva llave de cierre (línea **6**:) Un método no es más que un conjunto de instrucciones a las que se les asocia un nombre, de modo que si posteriormente se desea ejecutarlas basta referenciarlas por su nombre en vez de tener que reescribirlas.

La partícula que antecede al nombre del método indica cuál es el tipo de valor que se devuelve tras la ejecución del mismo, y en este caso es **void** que significa que no se devuelve nada. Por su parte, los paréntesis que se han colocado tras el nombre del método indican cuáles son los parámetros que este toma. Estos parámetros permiten variar el resultado de la ejecución del método en cada llamada al mismo, según los valores que para ellos se especifiquen. Como en este caso los paréntesis están vacíos, nuestro método no toma parámetros de ningún tipo.

Las palabras **public** y **static** que anteceden a la declaración del tipo de valor devuelto son modificadores opcionales del significado de la declaración de método. **Public** indica que el método es público; es decir, que puede llamársele desde código escrito dentro de cualquier otra clase. En caso de no incluirse este modificador se habría considerado que el método es **private**, lo que significa que solo sería correcto llamarle desde dentro de la misma clase en que se declara. Por su parte, **static** indica que es un método estático; es decir, asociado a la clase dentro de la que se define y no a los objetos que se creen a partir de la misma, por lo que para acceder a él se usará la sintaxis *nombreClase.NombreMétodo(parámetros)* –en nuestro caso *Lolo1.Main()*– y no *objeto.NombreMétodo(parámetros)* como corresponde a los métodos no estáticos.

El nombre, modificadores, tipos de parámetros y tipo de valor devuelto que se han dado al método *Main()* del ejemplo no son arbitrarios, sino que se corresponden con una de las cuatro posibles formas de definir el punto de entrada de nuestra aplicación. Este **punto de entrada** es sencillamente el método a partir del cual se comenzará a ejecutar el código de nuestra aplicación, y su declaración ha de ser de una de estas cuatro formas posibles:

public static int Main()

public static void Main()

public static void Main(String[] args)

public static int Main(String[] args)

El parámetro que puede tomar el método **Main()** almacena la lista de argumentos con los que se llamó a la aplicación; y como se ve, en caso de que no vayamos a utilizarla no es necesario especificarla en la declaración de Main(). El tipo de este parámetro es String[], que significa que es una tabla de cadenas de texto

(objetos String); y su nombre, que es el que habrá de usarse dentro del código del método Main() para hacerle referencia, puede ser cualquiera (en el ejemplo es args).

Por otro lado, la primera y última forma de uso del método Main() muestran que este no tiene porque no devolver ningún valor, sino que puede devolver uno de tipo int. Dicho valor sería interpretado como código de retorno de la aplicación, que suele usarse para indicar si la aplicación a terminado con éxito o no.

Finalmente, la única línea que nos queda por estudiar de nuestro sencillo programa de ejemplo es precisamente el código a ejecutar; es decir, el código de su método Main().

5: *System.Console.WriteLine("¡Hola Mundo!");*

Esta instrucción lo único que hace es ejecutar el método **WriteLine()** de la clase **Console.** Esta clase viene predefinida en la librería de clases de .NET, y **WriteLine()** es un método de clase (static) definido dentro de ella (al igual que nuestro **Main()** lo está en la clase Lolo1) cuyo código lo que hace es imprimir en el dispositivo de salida estándar de nuestra máquina (por defecto, la ventana de consola) la cadena de texto que le pasemos como parámetro. Una cadena de texto es cualquier secuencia de caracteres encerrada entre comillas dobles ("..."), aunque dichas comillas no forman parte de la cadena como puede observarse viendo que al ejecutar el ejemplo no se muestran en pantalla.

Antes de continuar es importante resaltar algunos aspectos:

1) C# es un lenguaje sensible a las mayúsculas, lo que significa que no da igual la capitalización con la que se debe escribir cada identificador. Es decir, no es lo mismo escribir *Console* que *COnsole* o que *CONSOLE*, y en caso de que lo hagamos de una de las dos últimas formas el compilador producirá un error indicando que no conoce ninguna clase con ese nombre. Un error común entre programadores acostumbrados a JAVA es llamar al punto de entrada del programa *main()* en vez de *Main().*

2) Todo el código escrito en un fichero de código fuente en C# es autocontenido. Es decir, no son necesarios ni ficheros de cabecera ni ficheros IDL ni ningún otro tipo de fichero adicional aparte del propio fichero de código fuente.

3) Por defecto, el compilador solo busca definiciones de clases predefinidas en el fichero **mscorlib.dll**, y si vamos a usar clases definidas en otro fichero hemos de indicárselo mediante la opción **/r** del compilador.

4) C# admite métodos y tipos genéricos, que proporcionan mayor rendimiento y seguridad de tipos, e iteradores, que permiten a los implementadores de clases de colección definir comportamientos de iteración personalizados que el código de cliente puede utilizar.

5) Una clase puede heredar directamente de una clase primaria, pero puede implementar cualquier número de interfaces. Los métodos

que reemplazan a los métodos virtuales en una clase primaria requieren la palabra clave **override** como medio para evitar redefiniciones accidentales. En C#, una estructura es como una clase sencilla; es un tipo asignado en la pila que puede implementar interfaces pero que no admite la herencia.

csc /t:winexe /r:System.Winforms.dll;System.dll;Microsoft.Win32.Interop.dll Fuente.cs

3. Espacios de nombres

Un espacio de nombres es una forma de organizar las clases definidas en la librería de .NET en grupos de clases relacionadas entre sí. Por ejemplo, dentro del espacio de nombres **System** usando en el ejemplo Lolo están incluidas todas las clases más frecuentemente usadas en cualquier aplicación .NET. Dado que puede ser tedioso tener que preceder todas nuestras referencias a clases con el nombre del espacio de nombres en que están definidas, en C# también se da la posibilidad de hacer:

```
using System;
class Lolo2     {
public static void Main()       {
                Console.WriteLine("¡Hola Mundo!");
        }
}
```

Las sentencias **using** siempre han de incluirse en el fichero fuente antes de la declaración de cualquier clase y permite indicar cuáles son los espacios de nombres que se usaran implícitamente. En nuestro ejemplo, gracias al uso esta sentencia no es necesario preceder la referencia a la clase **Console** con el nombre del espacio de nombres en que está definida, sino que el compilador automáticamente intentará encontrarla buscándola en el espacio de nombres **System**.

Los espacios de nombres también son útiles para evitar conflictos en caso de que se quieran usar clases de igual nombre pero procedentes de distintos fabricantes, pues las diferenciaríamos por su espacio de nombres. Para esto, es necesario que no coincidan los nombres de estos espacios, y una buena forma de hacerlo es dándoles el nombre de la empresa que desarrolló la clase, o el nombre del dominio de Internet de esta, etc.

Para indicar que una clase forma parte de un espacio de nombres basta incluir su definición dentro de la definición de un espacio de nombres. Por ejemplo, si queremos definir nuestra clase de ejemplo anterior dentro de un espacio de nombres llamado *Pruebas* bastaría añadir estas líneas:

```
using System;
namespace Pruebas   {
        class Lolo3  {
        public static void Main()       {
                        Console.WriteLine("¡Hola Mundo!");
        }
        }
}
```

4. Aplicaciones con argumentos

Antes se comentó que es posible declarar el método **Main()** de modo que tome un parámetro de tipo **String[]** que contenga los argumentos con los que se llamó a la aplicación. Es decir, de una de estas dos formas:

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

String[] indica que el parámetro **Main()** es una tabla de cadenas. Como en la mayoría de lenguajes de programación, una tabla no es más que una conjunto de valores ordenados y de tamaño fijo. Los corchetes **[]** en la declaración del parámetro indican que este es una tabla, y String indica que es una tabla de cadenas. Es importante resaltar el hecho de que aunque una tabla siempre tiene un tamaño fijo, este tamaño no forma parte de la declaración de la misma. Esto permite que una misma variable de tipo tabla pueda almacenar tablas de diferentes tamaños, aunque el tamaño de la tabla almacenada en cada instante no pueda modificarse.

Los elementos de la tabla de cadenas que puede tomar como parámetro el método **Main()** son cada uno de los argumentos con los que se llamó al programa. En C# las tablas se indexan desde 0, lo que significa que su primer argumento se almacena en la posición 0, el segundo en la posición 1, etc. Esto es importante tenerlo presente a la hora de acceder a cada elemento de una tabla, para lo que se usa la notación *tabla[posiciónElemento]* como muestra la siguiente variante de la clase *Lolo*.

```
using System;
class Lolo4     {
        public static void Main(String[] args)          {
                Console.WriteLine("¡Hola {0}!", args[0]);
        }
}
```

Es importante notar la forma especial en que se ha realizado la llamada al método **WriteLine().** En este caso, la cadena a imprimir contiene una secuencia de caracteres de la forma *{número}* que indica que se ha de mostrar en su lugar el valor del argumento número+2 de **WriteLine().** Así, en nuestro ejemplo *{0}* indica que se ha de mostrar el valor del segundo argumento; es decir, el de args[0], que es el primer argumento con que se llamó a *Lolo4* (en los arrays el indice empieza por 0).

Ahora cuando ejecutemos el programa hemos de pasarle un argumento y en función del valor que este tome se mostrará un mensaje de bienvenida personalizado. Por ejemplo, si ejecutamos el programa así: Lolo4 TAIER@S.

Se nos mostrará el siguiente mensaje de saludo: ¡Hola TAIER@S!

5. Tipos básicos

Tipo	Descripción	*Bi ts*	Rango	Alias
sbyte	*Bytes* con signo	8	-128 - 127	SByte
byte	*Bytes* con signo	8	0 - 255	Byte
short	Enteros cortos con signo	16	-32.768 - 32.767	Int16
ushort	Enteros cortos con signo	16	0 - 65.535	UInt16
int	Enteros normales	32	-2.147.483.648 - 2.147.483.647	Int32
uint	Enteros normales sin signo	32	0 - 4.294.967.295	UInt32
long	Enteros largos	64	-9.223.372.036.854.775.808	Int64
ulong	Enteros largos sin signo	64	0 - 18.446.744.073.709.551.615	Uint64
float	Reales con 7 dígitos de precisión	32	$1,5x10^{-45}$ - $3,4X10^{38}$	Float
double	Reales con 15-16 dígitos de precisión	64	$5,0x10^{-324}$ - $1,7X10^{308}$	Double
decimal	Reales con 28-29 dígitos de precisión	128	$1,0x10^{-28}$ - $7,9X10^{28}$	Decimal
oolean	Valores lógicos	32	true, false	Boolean
char	Caracteres Unicode	16	Unicode 0 – Unicode 65535	Char
string	Cadenas de caracteres	var	Permitido por memoria	String
object	Cualquier objeto	var	Depende del objeto	Object

En C# los tipos básicos son tipos del mismo nivel que cualquier otro tipo del lenguaje. Es decir, heredan de **System.Object** y pueden ser tratados como objetos de la misma por cualquier rutina que espere un **System.Object**, lo que cual es muy útil para el diseño de rutinas genéricas, que admitan parámetros de cualquier tipo. En realidad todos los tipos básicos de C# son simples alias de tipos del espacio de nombres **System**, como se recoge en la última columna de la tabla. Por ejemplo, **sbyte** es alias de **System.Sbyte** y da igual usar una forma del mismo u otra.

6. Instrucciones condicionales

C# ofrece una serie de instrucciones que permiten ejecutar bloques de código solo si se da una determinada condición. Estas son:

A) Intrucción If

Como la mayoría de los lenguajes de programación, C# incluye la instrucción condicional **if**, cuya forma de uso es:

```
if (condición)
    intruccionesIf
else
    intruccionesElse
```

El significado de esta instrucción es el siguiente: se evalúa la *condición* indicada, y en caso de ser cierta se ejecutan las *instruccionesIf*; mientras que si no lo es se ejecutan las *instruccionesElse*. La rama **else** es opcional y, si se omite y la condición es falsa, se seguiría ejecutando a partir de la siguiente instrucción al **if**. Si las *intruccionesIf* o las *instruccionesElse* constan de más de una instrucción, es necesario encerrar el conjunto de instrucciones de las que constan entre llaves (*{...}*)

Un ejemplo de aplicación de esta instrucción es esta variante del Lolo:

```
using System;


class Lolo5    {
        public static void Main(String[] args)        {
                if (args.Length > 0)
                        Console.WriteLine("¡Hola {0}!", args[0]);
                else
                        Console.WriteLine("¡Hola mundo!");
        }
}
```

En este caso, si ejecutamos el programa sin indicar ningún argumento al lanzarlo veremos que el mensaje que se imprime es *¡Hola Mundo!*, mientras que si lo ejecutamos indicando algún argumento se mostrará un mensaje de bienvenida personalizado (del mismo tipo que en el *Lolo2*).

Nótese que para saber si se han pasado argumentos en la llamada al programa y tomar una u otra decisión según el caso, lo que se hace en la condición del **if** es comprobar si la longitud de la tabla de argumentos es superior a 0. Para conocer esta longitud se utiliza la propiedad **Length** que toda tabla tiene definida. Recuérdese que el tamaño de una tabla es fijo, por lo que esta propiedad es de solo lectura y no es válido intentar escribir en ella (como por ejemplo, haciendo args.Length = 2).

B) Instrucción Switch

Para aquellos casos en los haya que ejecutar unos u otros bloques de instrucciones según el valor de una determinada expresión C# proporciona la instrucción condicional **switch**, cuya forma de uso es:

```
switch(condición)
{
        case caso1: instrucciones1: break;
        case caso2: instrucciones2: break;
        ...
        default: instruccionesDefecto; break;
}
```

El significado de esta instrucción es el siguiente: se evalúa la *condición*, y si su valor coincide con el de *caso1*, se ejecutan las *instruciones1*; si coincide con el de *caso2* se ejecutan las *instrucciones2*; y así para cada caso mientras no se encuentra alguno que coincida con el valor resultante de la evaluación. La rama **default** es opcional, y en caso de agotarse todos los casos y no encontrarse coincidencia, entonces se pasaría a ejecutar las *instruccionesDefecto* en caso de que dicha rama apareciese; y si no apareciese, se pasaría directamente a ejecutar la instrucción siguiente al **switch.**

En realidad, la rama **default**, si se usa, no tiene por qué aparecer la última, aunque se recomienda que lo haga porque ello facilita la legibilidad del código.

Un ejemplo de uso de esta instrucción es el siguiente:

```
using System;
class Lolo6   {
   public static void Main(String[] args)      {
     if (args.Length > 0)
       switch(args[0])       {
        case "Oposiciones": Console.WriteLine("Hola, estas en oposiciones.");
             break;
        case "General": Console.WriteLine("Hola, estas en un curso estándar.");
             break;
        default: Console.WriteLine("Hola {0}", args[0]);
       }
     else
      Console.WriteLine("Hola Mundo");
 }
   }
```

Ahora, nuestro programa reconoce a algunas personas y las saluda de forma especial. Nótese que al final de cada grupo de instrucciones se ha de incluir una instrucción **break** que indique el final de la lista de instrucciones asociadas a esa rama del **switch**. Esta instrucción puede ser sustituida por una instrucción **goto** usada de la forma *goto casoi* (o *goto default*) que indique qué otras ramas del **switch** han de ejecutarse tras llegar a ella. Además, en la última rama del **switch** no tiene por qué aparecer obligatoriamente ninguna de estas dos sentencias.

Para los programadores habituados a lenguajes como C++ es importante resaltarles el hecho de que, a diferencia de dicho lenguaje, C# obliga a incluir una sentencia **break** o una sentencia **goto** al final de cada rama del **switch**, con la idea de evitar errores muy comunes en este lenguaje, donde no es forzoso hacerlo.

7. Instrucciones iterativas

C# ofrece un buen número de instrucciones que permiten la ejecución de bloques de códigos repetidas veces. A continuación se comentan las principales:

A) Instrucción While

Es la instrucción iterativa más común en los lenguajes de programación, y en C# se usa de la siguiente forma:

```
while (condición)
  instrucciones
```

El significado de esta instrucción es el siguiente: se evalúa la *condición* y, en caso de ser cierta, se ejecutan las *instrucciones*. Tras ejecutarlas, se repite el proceso de evaluar la *condición* y ejecutar las *instrucciones* en caso de seguir siendo cierta. Este proceso se repite continuamente hasta que la condición deje de verificarse. Si las *instrucciones* constan de más de una instrucción es necesario encerrarlas entre llaves, del mismo modo que se comentó para el caso del **if**.

Un ejemplo de aplicación de esta sentencia es:

```
using System;
class Lolo7    {
        public static void Main(String[] args)                {
        int actual = 0;

        if (args.Length > 0)
                while (actual < args.Length)                  {
                        Console.WriteLine("¡Hola {0}!", args[actual]);
                        actual  = actual + 1;
                }
        else
                Console.WriteLine("¡Hola mundo!");
    }
}
```

En este caso, si se indica más de un argumento en la llamada a nuestro programa se mostrará por pantalla un mensaje de saludo para cada uno de ellos.

Observa que la primera línea del método **Main()** no contiene ahora una instrucción, sino que contiene una declaración de una variable de tipo **int**, nombre *actual* y valor inicial o que usaremos en la sentencia iterativa para saber cuál es la posición del argumento a mostrar en cada ejecución de la misma. El valor de esta variable se irá aumentando en una unidad, para así asegurar que siempre mantiene el valor adecuado para ir mostrando cada uno de los argumentos de llamada y para asegurar que la instrucción **while** termine de ejecutarse alguna vez, lo cual ocurrirá cuando se hallan mostrado todos los argumentos.

C# no proporciona ningún valor inicial a las variables locales de los métodos, por lo que es tarea del programador proporcionárselos antes de ser leídos. En cualquier caso, si el compilador detecta que en el código hay alguna posibilidad de que se lea algún parámetro no inicializado informará al programador de ello dando error. La idea detrás de todo esto es conseguir evitar errores comunes y difíciles de detectar que se dan en otros lenguajes cuando se olvida inicializar un parámetro y su valor por defecto no es el esperado.

B) Instrucción Do...while

Es una variante del while que se usa así:

```
do
  instrucciones
while (condición);
```

En este caso, el significado de la instrucción es ahora el siguiente: se ejecutan las *instrucciones* (que en caso de ser varias habrán de ir encerradas entre llaves), y tras ello se evalúa la condición. Si el resultado de evaluarla es cierto se vuelve a repetir el proceso, mientras que si no lo es se continúa ejecutando a partir de la instrucción siguiente al **do ... while.** Es importante resaltar que en esta instrucción es obligatorio incluir el punto y coma (;) al final del paréntesis de cierre de la condición, ya es un error frecuente entre novatos olvidar incluirlos.

Do ... while especialmente útil para aquellos casos en los que hay que asegurar que las instrucciones en él contenidas se ejecuten al menos una vez, aun cuando la condición sea falsa desde el principio. Un ejemplo de su uso es este código:

```
using System;
class Lolo8 {
        public static void Main()       {
                String leído;

                do      {
                        Console.WriteLine("Clave: ");
                        leído = Console.ReadLine();
                }
                while (leído != "Donald");
                Console.WriteLine("Hola Donald");
        }
}
```

Al ejecutarse la aplicación puede observarse que lo que se hace es preguntar al usuario una clave, de modo que mientras no se introduzca la clave correcta (que es *Donald*), no se continuará con la ejecución de la aplicación; y una vez que se introduzca correctamente se dará un mensaje de bienvenida al usuario.

El método **ReadLine()** de la clase **Console** detiene la ejecución de la aplicación y la deja en espera de que el usuario introduzca una cadena de caracteres y pulse la tecla ENTER, cadena que es devuelta por el método **ReadLi-**

ne() y que en la aplicación se guarda en la variable de tipo cadena llamada *leído* para comprobar posteriormente, en la condición del **do ... while**, si coincide con la clave esperada (*Donald*).

C) Instrucción For

Es otra variante del **while** que permite compactar el código de este tipo de bloques. Su forma de uso es:

```
for (inicialización; condición; incremento)
  instrucciones
```

El significado de esta instrucción es el siguiente: se realizan las inicializaciones de variables indicadas en *inicialización* y luego se evalúa la *condición*; si es cierta, se ejecutan las *instrucciones* indicadas (entre llaves si son varias). Tras ello se ejecutan las operaciones de *incremento* (o decremento) indicadas y se reevalúa la condición. Mientras esta sea cierta se irá repitiendo el proceso de ejecución de *instrucciones, incremento* de variables y reevaluación de la condición hasta que deje de serlo. En caso de que se desee inicializar o declarar varias variables en el campo de *inicialización* o de que se realizar varias operaciones incremento/decremento en el campo decremento habría que separarlas mediante comas (,).

Como se ve, la instrucción **for** recoge de una forma muy compacta el uso principal de la instrucción **while** normal, siendo un ejemplo de su uso:

```
using System;
class Lolo9    {
        public static void Main(String[] args)        {
                if (args.Length > 0)
                        for (int actual = 0; actual < args.Length; actual++)
                                Console.WriteLine("¡Hola {0}!", args[actual]);
                else
                        Console.WriteLine("¡Hola mundo!");
}       }
```

El funcionamiento de este ejemplo es exactamente el mismo que el del *Lolo5*, solo que en este caso se ha aprovechado la eficacia de la instrucción for para hacer reducir mucho más el tamaño del código. Si acaso, cabría señalar la utilización del operador ++ en el campo de incremento, cuyo significado es sumar 1 a la variable sobre la que se aplica. Simétricamente, también está definido el operador —, cuyo significado es restar 1 a la variable sobre la que es aplicado.

D) Instrucción Foreach

Esta instrucción es la novedad más importante introducida en el juego de instrucciones de C# respecto a JAVA y C++, sus más directos competidores. Se utiliza así:

foreach (tipoElemento elemento in colección)

instrucciones

Esta instrucción se utiliza para recorrer colecciones de elementos (por ejemplo, tablas), y su significado es muy sencillo: se ejecutan las *intrucciones* indicadas (estarán encerradas entre paréntesis en caso de ser varias) para cada uno de los elementos de la colección que se especifica.

El siguiente ejemplo muestra cómo se utiliza esta instrucción:

```
using System;
class Lolo10   {
        public static void Main(String[] args)                  {
                if (args.Length > 0)
                        foreach(String arg in args)
                                Console.WriteLine("¡Hola {0}!", arg);
                else
                        Console.WriteLine("¡Hola mundo!");
}       }
```

El significado de este ejemplo es el mismo que el del Lolo5.

8. Ejemplos

- **Clases y objetos**

```
class Lolo
{
    public Lolo(string dat1, string dat2, string dat3, string dat4)
    {
        this.Dat1=dat1;
        this.Dat2=dat2;
        this.Dat3=dat3;
        this.Dat4=dat4;
    }
```

```
    public double Metodo1
    {
        get
            {
                return this.metodo1;
            }
    }

    protected double metodo1=0;
    public string Dat1;
    public string Dat2;
     public string Dat3;
     public string Dat4;

     public void Metodo2(double argum)
     {
       Console.WriteLine("{0} ", argum);
        this.metodo1 += argum;
    }

     public void Metodo3(double argum)
     {
         Console.WriteLine("{0} ", argum);
    }

     public void Metodo4(double argum)
     {
         Console.WriteLine("{0} ", argum);
          this.metodo1 -= argum;
    }
}
```

```
class Ejemplo
{
    static void Main()
     {
          Lolo MiLolo=new Lolo("TAI", "2008", "ADAMS","100%Aprobados");

          Console.WriteLine("Los datos de mi Lolo son:");
          Console.WriteLine("Dat1: {0}", MiLolo.Dat1);
          Console.WriteLine("Dat2: {0}", MiLolo.Dat2);
          Console.WriteLine("Dat3: {0}", MiLolo.Dat3);
          Console.WriteLine("Número de bastidor: {0}", MiLolo.Dat4);

          MiLolo.Metodo2(100);
          Console.WriteLine("{0} ",MiLolo.Metodo1);
          MiLolo.Metodo4(75);
          Console.WriteLine("{0} ",MiLolo.Metodo1);

          MiLolo.Metodo3(45);

    }
}
```

C# no soporta herencia múltiple de clases. Una clase puede derivarse de otra, pero no de varias. Sí se puede derivar de otra clase y varias "interfaces".

```
class LoloAvanzado:Lolo
{
    public LoloAvanzado(string dat1, string dat2, string dat3, string dat4):
base(dat1, dat2, dat3, dat4) {}

    public override void Metodo2(double argum)
     {
          Console.WriteLine("{0}", argum);
           this.metodo1 += argum;
    }
}
```

```
LoloAvanzado MiLolo;
...
MiLolo = new LoloAvanzado("TAI", "2008", "ADAMS", "100%Aprobados ");
MiLolo.Metodo2(100);
```

Declarar una variable o un puntero en C# se escribe igual que en C:

```
int a;
int* punt;
```

- **C# soporta la sobrecarga de métodos**

```
namespace Espace
{
    // Aquí van las clases del espacio de nombres
}
```

```
using Espace.Tipo.Terrestre;
...
CLASE1 objeto = new CLASE1 (argumentos);

internal class NombreClase{
    // miembros de la clase
}
```

Si quieres que una clase sea accesible desde otros ensamblados, necesitarás que sea pública, usando el modificador de acceso public:

```
public class NombreClase
{
     // Aquí se codifican los miembros de la clase
}
```

Los indicadores, al igual que las clases, también tienen modificadores de acceso. Si se pone, ha de colocarse en primer lugar. Si no se pone, el compilador entenderá que es private. Dichos modificadores son:

MODIFICADOR	COMPORTAMIENTO
public	Hace que el indicador sea accesible desde otras clases.
protected	Hace que el indicador sea accesible desde otras clases derivadas de aquella en la que está declarado, pero no desde el cliente.
private	Hace que el indicador solo sea accesible desde la clase donde está declarado.
internal	Hace que el indicador solo sea accesible por los miembros del ensamblaje actual.

Para declarar una variable de uno de estos tipos en C# hay que colocar primero el tipo del CTS o bien el alias que le corresponde en C#, después el nombre de la variable y después, opcionalmente, asignarle su valor:

```
System.Int32 num=10;
int num=10;
```

La variable num sería de la clase System.Int32 en ambos casos: en el primero hemos usado el nombre de la clase tal y como está en el CTS, y en el segundo hemos usado el alias para C#. En todos los lenguajes que cumplen las especificaciones del CLS se usan los mismos tipos de datos, es decir, los tipos del CTS, aunque cada lenguaje tiene sus alias específicos.

- **Conversiones**

```
int a = System.Int32.Parse(cadena);
```

- **Métodos**

```
using System;

namespace HolaMundo
{
    class HolaMundoApp
    {
        static void Main(string[] args)
        {
```

```
                Console.WriteLine("Hola Mundo");
                string a = Console.ReadLine();
            }
        }
}
```

acceso tipo NombreMetodo(**TipoArg1 arguento1, TipoArg2 arguento2 ...**)

```
{
    // implementación del método
}
using System;
  class Clase2
{
    protected int dat3=0;
    protected byte dat4=100;

    public bool Metodo4(byte argum)
    {
        if (argum>this.dat4) return false;

            return this.dat4;
        }
    }
}
```

- **Sobrecarga de métodos**

*public bool Metodo(**single** cant)*

{...}

*public int Metodo(double argum, **double argumento2**)*

{...}

*public int Metodo(**single** argum, **double argumento2**)*

{...}

- **Métodos static**

```
using System;

namespace Estado
{
    class Estado
    {
        public static ushort Num_aprob()
        {
            return 15;
        }

        // miembros de la clase
    }

    class EstadoApp
    {
        static void Main()
        {
            Console.WriteLine("{0}", Estado.Num_aprob());

            string a=Console.ReadLine();
        }
    }
}
```

- **Constructores**

```
using System;
 namespace Constructores
{
    class Objeto
    {
        public Objeto()
        {
            Console.WriteLine("Instanciado el objeto");
        }
    }
```

```
    class ConstructoresApp
    {
        static void Main()
        {
            Objeto o = new Objeto();
            string a=Console.ReadLine();
        }
    }
}
namespace ConstructoresStatic
{
    class Mensaje
    {
        public static string Texto;

        static Mensaje()
        {
            Texto=”Hola “;
        }
    }

    class ConstructoresStaticApp
    {
        static void Main()
        {
            Console.WriteLine(Mensaje.Texto);

            string a=Console.ReadLine();
        }
    }
}
```