PROGRAMACIÓN EN C#

Aspectos léxicos

Comentarios

Un comentario es texto cuyo contenido es, por defecto, completamente ignorado por el compilador.

En C# hay dos formas de escribir comentarios.

1. La primera consiste en encerrar todo el texto que se desee (incluyendo varias líneas) comentar entre caracteres /* y */ siguiendo la siguiente sintaxis:

   /*<texto>*/

   No es posible anidar comentarios de este tipo.

2. C# ofrece una sintaxis alternativa más compacta para la escritura este tipo de comentarios en las que se considera como indicador del comienzo del comentario la pareja de caracteres // y como indicador de su final el fin de línea. Por tanto, la sintaxis que siguen estos comentarios es:

   // <texto>

   Pueden anidarse sin ningún problema.

3. También está la posibilidad de la triple /// que lo que hace es incorporar ayuda al Visual Studio.

La guía de estilo de la Universidad de Alicante determina que:

- No se debe escribir comentario par cada línea de código y variable declarada.
- Solo se escriben comentarios cuando hacen falta.
- Un código bien escrito y legible requiere de pocos comentarios. Se entiende por código legible a un código con nombres de variables y métodos legibles.
- Pocas líneas de comentario hacen el código legible.
- Un código complejo o extraño llevará suficientes comentarios.
- Una inicialización de una variable numérica con un valor especial (dtto de 0, -1) necesita de una razón.
- Cuida la ortografía, gramática y puntuación.

Identificadores

Al igual que en cualquier lenguaje de programación, en C# un identificador no es más que, como su propio nombre indica, un nombre con el que identificaremos algún elemento de nuestro código, ya sea una clase, una variable, un método, etc.

Típicamente el nombre de un identificador será una secuencia de cualquier número de caracteres alfanuméricos -incluidas vocales acentuadas y eñes- tales que el primero de ellos no sea un número. Por ejemplo, identificadores válidos serían: Arriba, caña, C3P0, áëÎò, etc; pero no lo serían 3com, 127, etc.

Sin embargo, y aunque por motivos de legibilidad del código no se recomienda. C# también permite incluir dentro de un identificador caracteres especiales imprimibles tales como símbolos de diéresis, subrayados, etc. siempre y cuando estos no tengan un significado especial dentro del lenguaje. Por ejemplo, también serían identificadores válidos _barco_, c¨k y A·B pero no C# (# indica inicio de directiva de preprocesado) o a!b (! indica operación lógica "not")

La guía de estilo de la Universidad de Alicante determina que:
Los identificadores deben legibles y deben ser consecuentes con lo que identifica. Por ello se evitarán diminutivos del tipo id, varnum, etc.

Palabras reservadas

Aunque antes se han dado una serie de restricciones sobre cuáles son los nombres válidos que se pueden dar en C# a los identificadores, falta todavía por dar una: los siguientes nombres no son válidos como identificadores ya que tienen un significado especial en el lenguaje:

abstract, as, base, bool, break, byte, case, catch, char, checked, class, const, continue, decimal, default, delegate, do, double, else, enum, event, explicit, extern, false, finally, fixed, float, for, foreach, goto, if, implicit, in, int, interface, internal, lock, is, long, namespace, new, null, object, operator, out, override, params, private, protected, public, readonly, ref, return, sbyte, sealed, short, sizeof, stackalloc, static, string, struct, switch, this, throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort, using, virtual, void, while

Aparte de estas palabras reservadas, si en futuras implementaciones del lenguaje se decidiese incluir nuevas palabras reservadas, Microsoft dice que dichas palabras habrían de incluir al menos dos símbolos de subrayado consecutivos (__) Por tanto, para evitar posibles conflictos futuros no se recomienda dar a nuestros identificadores nombres que contengan dicha secuencia de símbolos.

Operadores

Un operador en C# es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

- Operaciones aritméticas: Los operadores aritméticos incluidos en C# son los típicos de suma (+), resta (-), producto (*), división (/) y módulo (%). También se incluyen operadores de "menos unario" (-) y "más unario" (+)

```
using System;
using escribe = System.Web.HttpContext;
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string texto;
        texto = ((+5) + "<br />");          // suma unarias
        texto +=  ((5 + 5) + "<br />");      // suma
        texto +=  ((5 + .5) + "<br />");     // suma
        texto += ("5" + "5") + "<br />"; // concatena string
        texto += (5.0 + "5") + "<br />"; // concatena string
        escribe.Current.Response.Write (texto);// Conversión automática de double a
string
    }
}
```

- Operaciones lógicas: Se incluyen operadores que permiten realizar las operaciones lógicas típicas: "and" (&& y &), "or" (|| y |), "not" (!) y "xor" (^)

  Los operadores && y || se diferencia de & y | en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación no perezosa consiste en evaluar siempre ambos operandos. Es decir, si el primer operando de una operación && es falso se devuelve false directamente, sin evaluar el segundo; y si el primer operando de una || es cierto se devuelve true directamente, sin evaluar el otro.

- Operaciones relacionales: Se han incluido los tradicionales operadores de igualdad (==), desigualdad (!=), "mayor que" (>), "menor que" (<), "mayor o igual que" (>=) y "menor o igual que" (<=)

- Operaciones de manipulación de bits: Se han incluido operadores que permiten realizar a nivel de bits operaciones "and" (&), "or" (|), "not" (~), "xor" (^), desplazamiento a izquierda (<<) y desplazamiento a derecha (>>) El operador << desplaza a izquierda rellenando con ceros, mientras que el tipo de relleno realizado por >> depende del tipo de dato sobre el que se aplica: si es un dato con signo mantiene el signo, y en caso contrario rellena con ceros.

- Operaciones de asignación: Para realizar asignaciones se usa en C# el operador =, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión a = b asigna a la variable a el valor de la variable b y devuelve dicho valor, mientras que la expresión c = a = b asigna a **c** y **a** el valor de b (el operador = es asociativo por la derecha)

  También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:

  ```
  temperatura = temperatura + 15;   // Sin usar asignación compuesta
  temperatura += 15;                // Usando asignación compuesta
  ```

  Las dos líneas anteriores son equivalentes.

Aparte del operador de asignación compuesto +=, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: +=, -=, *=, /=, %=, &=, |=, ^=, <<= y >>=. Nótese que no hay versiones compuestas para los operadores binarios && y ||.

Otros dos operadores de asignación incluidos son los de incremento(++) y decremento (--) Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

```
temperatura = temperatura + 1;  // Sin asignación compuesta ni incremento
temperatura += 1;               // Usando asignación compuesta
temperatura++;                  // Usando incremento
```

Si el operador ++ se coloca:

```
c = b++; // Se asigna a c el valor de b y luego se incrementa b
c = ++b; // Se incrementa el valor de b y luego se asigna a c
```

- Operaciones con cadenas: (+) "Hola"+" mundo" devuelve "Hola mundo". Se deben evitar. Para concatenar cadenas de texto muy grandes no utilizar ni + ni 'concat'. Mucho mejor utilizar el objeto StringBuider. La velocidad de ejecución es muuuucho mayor.

```
StringBuilder sSql = new StringBuilder("SELECT  nombre, apellidos");

sSql.Append(" , (select cosas from cosas_importantes");
sSql.Append("  where cosas_importantes.dni = personas.dni) ");
sSql.Append(" FROM personas ");
```

Ejemplo:

```
using System;
using System.Text;
using System.Web;

public partial class Default4 : System.Web.UI.Page
{

    protected void Page_Load(object sender, EventArgs e)
    {
            // Concatenation using string
            HttpContext.Current.Response.Write("<hr />Rutina string");
            string str = string.Empty;
            DateTime startTime = DateTime.Now;
            HttpContext.Current.Response.Write("<hr>Incio:" + startTime.ToString());
            for (int i = 0; i < 100000; i++)
            {
                str += i.ToString();
            }
            //HttpContext.Current.Response.Write("Cadena: " + str);
            DateTime stopTime = DateTime.Now;
            HttpContext.Current.Response.Write("<br />Fin:" + stopTime.ToString());
            // Concatenation using StringBuilder
            HttpContext.Current.Response.Write("<hr />StringBuilder");
            StringBuilder builder = new StringBuilder();
            startTime = DateTime.Now;
            HttpContext.Current.Response.Write("<br />Inicio:" + startTime.ToString());
            for (int i = 0; i < 100000; i++)
            {
                builder.Append(i.ToString());
            }
            //HttpContext.Current.Response.Write("Cadena: " + builder);
```

```
            stopTime = DateTime.Now;
            HttpContext.Current.Response.Write("<br />Fin:" + stopTime.ToString());
        }
}
```

Resultado:

Rutina string

Incio:09/09/2010 19:10:06
Fin:09/09/2010 19:11:25

StringBuilder
Inicio:09/09/2010 19:11:25
Fin:09/09/2010 19:11:25

- Operaciones de acceso a arrays (matrices): Una array es un conjunto ordenado de objetos de tamaño fijo. Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo [] sobre el array para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder dentro del conjunto.

  <array>[<posiciónElemento>];

- Operador condicional: Es el único operador incluido en C# que toma 3 operandos, y se usa así:

  <condición> ? <expresión1> : <expresión2>

  El significado del operando es el siguiente: se evalúa <condición> Si es cierta se devuelve el resultado de evaluar <expresión1>, y si es falsa se devuelve el resultado de evaluar <condición2>.

  b = (a>0)? a : 0; // Suponemos a y b de tipos enteros

  En este ejemplo, si el valor de la variable a es superior a 0 se asignará a b el valor de a, mientras que en caso contrario el valor que se le asignará será 0.

  Hay que tener en cuenta que este operador es asociativo por la derecha, por lo que una expresión como a?b:c?d:e es equivalente a a?b:(c?d:e)

  No hay que confundir este operador con la instrucción condicional if, pues aunque su utilidad es similar al de ésta, ? devuelve un valor e if no.

- Operaciones de acceso a objetos: Para acceder a los miembros de un objeto se usa el operador **.**, cuya sintaxis es:

  <objeto>.<miembro>

  Si a es un objeto, ejemplos de cómo llamar a diferentes miembros suyos son:

  a.b = 2; // Asignamos a su propiedad a el valor 2
  a.f(); // Llamamos a su método f()

```
a.g(2);   // Llamamos a su método g() pasándole como parámetro el valor entero 2
a.c += new adelegado(h)     // Asociamos a su evento c el código del método h() de
                            // "tipo" adelegado
```

- Operaciones de obtención de información sobre tipos:

  typeof(<nombreTipo>)

  Este operador devuelve un objeto de tipo System.Type con información sobre el tipo de nombre <nombreTipo> que podremos consultar a través de los miembros ofrecidos por dicho objeto. Esta información incluye detalles tales como cuáles son sus miembros, cuál es su tipo padre o a qué espacio de nombres pertenece.

  Si lo que queremos es determinar si una determinada expresión es de un tipo u otro, entonces el operador a usar es is, cuya sintaxis es la siguiente:

  <expresión> is <nombreTipo>

- Otro operador que permite obtener información sobre un tipo de dato: sizeof Este operador permite obtener el número de bytes que ocuparán en memoria los objetos de un tipo, y se usa así:

  sizeof(<nombreTipo>)

- Operaciones de **creación de objetos**: El operador más típicamente usado para crear objetos es new, que se usa así:

  new <nombreTipo>(<parametros>)

  Este operador crea un objeto de <nombreTipo> pasándole a su método constructor los parámetros indicados en <parámetros> y devuelve una referencia al mismo.

  Se pueden tener varios constructores que se diferencien en el número de parámetros. Así según el tipo y número de estos parámetros se llamará a uno u otro. Así, suponiendo que a1 y a2 sean variables de tipo Avión, ejemplos de uso del operador new son:

  ```
  Avión a1 = new Avión(); // Se llama al constructor sin parámetros de Avión
  Avión a2  = new Avión("Caza"); // Se llama al constructor de Avión que toma
                                 // como parámetro una cadena
  ```

- **Operaciones de conversión (cast)**:
  Para convertir unos objetos en otros se utiliza el operador de conversión, que no consiste más que en preceder la expresión a convertir del nombre entre paréntesis del tipo al que se desea convertir el resultado de evaluarla. Por ejemplo, si l es una variable de tipo long y se desea almacenar su valor dentro de una variable de tipo int llamada i, habría que convertir previamente su valor a tipo int así:

```
i = (int) l;  // Asignamos a i el resultado de convertir el valor de l a tipo int
```

Los tipos int y long están predefinidos en C# y permite almacenar valores enteros con signo. La capacidad de int es de 32 bits, mientras que la de long es de 64 bits. Por tanto, a no ser que hagamos uso del operador de conversión, el compilador no nos dejará hacer la asignación, ya que al ser mayor la capacidad de los long, no todo valor que se pueda almacenar en un long tiene porqué poderse almacenar en un int. Es decir, no es válido:

```
i = l; //ERROR: El valor de l no tiene porqué caber en i
```

Esta restricción en la asignación la impone el compilador debido a que sin ella podrían producirse errores muy difíciles de detectar ante truncamientos no esperados debido al que el valor de la variable fuente es superior a la capacidad de la variable destino.

En caso de que se solicite hacer una conversión inválida devuelve una excepción System.InvalidCastException.

Tipos de datos:

C#	Tipo en System	Características	Símbolo
sbyte	System.Sbyte	entero, 1 byte con signo	
byte	System.Byte	entero, 1 byte sin signo	
short	System.Short	entero, 2 bytes con signo	
ushort	System.UShort	entero, 2 bytes sin signo	
int	System.Int32	entero, 4 bytes con signo	
uint	System.UInt32	entero, 4 bytes sin signo	U
long	System.Int64	entero, 8 bytes con signo	L
ulong	System.ULong64	entero, 8 bytes sin signo	UL
float	System.Single	real, IEEE 754, 32 bits	F
double	System.Double	real, IEEE 754, 64 bits	D
decimal	System.Decimal	real, 128 bits (28 dígitos significativos)	M
bool	System.Boolean	(Verdad/Falso) 1 byte	
char	System.Char	Carácter Unicode, 2 bytes	´ ´
string	System.String	Cadenas de caracteres Unicode	" "
object	System.Object	Cualquier objeto (ningún tipo concreto)	

El valor que **por defecto** se da a los campos de tipos básicos consiste en poner a cero todo el área de memoria que ocupen. Esto se traduce en que los campos de tipos básicos numéricos se inicializan por defecto con el valor 0, los de tipo bool lo hacen con false, los de tipo char con '\u0000', y los de tipo string y object con null.

La guía de estilo de la Universidad de Alicante determina que:

Se deben usar los tipos especificos de C#, en vez de los alias definidos en el espacio de nombres System.

Bien:	**Mal**:
int age; string name; object contactInfo;	Int16 age; String name; Object contactInfo;

Conversión entre tipos de datos

Tipo numérico	**Método**		**Tipo numérico**	**Método**
decimal	ToDecimal(String)		long	ToInt64(String)
float	ToSingle(String)		ushort	ToUInt16(String)
double	ToDouble(String)		uint	ToUInt32(String)
short	ToInt16(String)		ulong	ToUInt64(String)

Ejemplo
Este ejemplo llama al método ToInt32(String) para convertir la cadena "29" en int. Después, suma 1 al resultado e imprime la salida.

```
int numVal = Convert.ToInt32("29");
numVal++;
Console.WriteLine(numVal);
// Output: 30
```

Otra forma de convertir string en int es mediante los métodos Parse o TryParse de la estructura System.Int32. El método ToUInt32 utiliza Parse internamente. Si el formato de la cadena no es válido, Parse produce una excepción, mientras que TryParse no produce ninguna excepción, pero devuelve false. En los siguientes ejemplos se muestran llamadas correctas e incorrectas a Parse y TryParse.

```
int numVal = Int32.Parse("-105");
Console.WriteLine(numVal);
// Output: -105
```

```
int j;
Int32.TryParse("-105", out j);
Console.WriteLine(j);
// Output: -105
```

```
try
{
    int m = Int32.Parse("abc");
}
catch (FormatException e)
{
    Console.WriteLine(e.Message);
```

```
}
// Output: Input string was not in a correct format.
```

```
string inputString = "abc";
int numValue;
bool parsed = Int32.TryParse(inputString, out numValue);

if (!parsed)
    Console.WriteLine("Int32.TryParse could not parse '{0}' to an int.\n", inputString);

// Output: Int32.TryParse could not parse 'abc' to an int.
```

En caso de una fecha tendremos:

```
DateTime.TryParse(FechaNac, out this.fechanac);
DateTime. Parse(FechaCurso);
```

Ejercicio 1:

A partir de una caja de texto y un botón de confirmar sacar un mensaje en pantalla que determine si el dato que se introdujo es:

- Un número positivo o negativo
- Un número par o impar
- Un número con decimales o sin ellos.
- Una fecha
- Un texto

Clases

Conceptos de clase y objeto

C# es un lenguaje orientado a objetos puro, lo que significa que todo con lo que vamos a trabajar en este lenguaje son objetos. Un objeto es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa en C# no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

Una clase es la definición de las características concretas de un determinado tipo de objetos. Es decir, de cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo. Por esta razón, se suele decir que el tipo de dato de un objeto es la clase que define las características del mismo.

Sintaxis de definición de clases

class <nombreClase> { <miembros> }

Según la guía de estilo del CPD de la Universidad de Alicante se requiere que el nombre de la clase siga una notación Pascal (ej: HolaMundo, Taxi).

Para indentar se usa TAB y no los espacios.

Las llaves deben estar en el mismo nivel que el código fuera de las llaves.

Se debe usar una línea en blanco para separar los grupos lógicos de código.

Los miembros de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma.

Aunque en C# hay muchos tipos de miembros distintos, por ahora vamos a considerar que estos únicamente pueden ser campos o métodos y vamos a hablar un poco acerca de ellos y de cómo se definen:

- **Campos**: Un campo es un dato común a todos los objetos de una determinada clase. Se trata en realidad de un variable definida dentro de una definición de clase. Para definir cuáles son los campos de los que una clase dispone se usa la siguiente sintaxis dentro de la zona señalada como <miembros> en la definición de la misma:

  <tipoCampo> <nombreCampo>;

  Según la guía de estilo del CPD de la Universidad de Alicante se requiere que el nombre de las variables, campos y parámetros siga una notación Camel (ej: holaMundo, taxi).

  Evitar los subrayados.

  Usar palabras significativas para nombrar las variables:

  * No usar abreviaciones Usar nombre, dirección en vez de nom, dir, etc.
  * No usar nombres de variables de un solo carácter para nombres de variables Usar nombres como índice o temporal.

  Una excepción es el caso de las variables que se utilizan para bucles:

  ```
  for ( int i = 0; i < count; i++ )
  {
  ...
  }
  ```

  Si una variable sólo se utiliza como contador en una iteración y no se usa más, se puede poner una variable de un sólo caracter (i) en vez de inventar otro nombre más apropiado.

Un campo debe ser privado si seguimos las buenas prácticas de programación. En este caso: Notación Camel con un subrayado al principio del nombre

```
private int _recordId ;
```

Los campos de un objeto son a su vez objetos, y en <tipoCampo> hemos de indicar cuál es el tipo de dato del objeto que vamos a crear. A continuación se muestra un ejemplo de definición de una clase de nombre Persona que dispone de tres campos:

```
class Persona
{
  private string _nombre;   // Campo de cada objeto Persona que almacena su nombre
  prívate int _edad;          // Campo de cada objeto Persona que almacena su edad
}
```

Según esta definición, todos los objetos de clase Persona incorporarán campos que almacenarán cuál es el nombre de la persona que cada objeto representa y cuál es su edad.

Para acceder a un campo de un determinado objeto se usa la sintaxis:

<objeto>.<campo>

Por ejemplo, para acceder al campo Edad de un objeto Persona llamado p y cambiar su valor por 20 se haría: p.Edad = 20;

- **Métodos**: Un método es un conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas basta referenciarlas a través de dicho nombre en vez de tener que escribirlas. Dentro de estas instrucciones es posible acceder con total libertad a la información almacenada en los campos pertenecientes a la clase dentro de la que el método se ha definido: los métodos permiten manipular los datos almacenados en los objetos.

  La sintaxis que se usa en C# para definir los métodos es la siguiente:

  <tipoDevuelto> <nombreMétodo> (<parametros>) { <instrucciones> }

Según la guía de estilo del CPD de la Universidad de Alicante se requiere que el nombre de las variables, campos y parámetros siga una notación Camel (ej: holaMundo, taxi). Ver página anterior.

La notación de los métodos debe ser Pascal (ej: HolaMundo), al igual que el nombre de las clases.

Todo método puede devolver un objeto como resultado de la ejecución de las instrucciones que lo forman, y el tipo de dato al que pertenece este objeto es lo que se indica en <tipoDevuelto>.

Si no devuelve nada se indica void, y si devuelve algo es obligatorio finalizar la ejecución de sus instrucciones con alguna instrucción return <objeto>; que indique qué objeto ha de devolverse.

Opcionalmente todo método puede recibir en cada llamada una lista de objetos a los que podrá acceder durante la ejecución de sus instrucciones. En <parametros> se indica es cuáles son los tipos de dato de estos objetos y cuál es el nombre con el que harán referencia las instrucciones del método a cada uno de ellos. Aunque los objetos que puede recibir el método pueden ser diferentes cada vez que se solicite su ejecución, siempre han de ser de los mismos tipos y han de seguir el orden establecido en <parametros>.

Un ejemplo de cómo declarar un método de nombre Cumpleaños es la siguiente modificación de la definición de la clase Persona usada antes como ejemplo:

```
class Persona   {
  private string _nombre;       // Campo de cada objeto Persona almacena nombre
  private int _edad;            // Campo de cada objeto Persona almacena edad

  // Incrementa en uno de la edad del objeto Persona
  void Cumpleaños()      {      edad++;      }
}
```

La sintaxis usada para llamar a los métodos de un objeto es la misma que la usada para llamar a sus campos, sólo que ahora tras el nombre del método al que se desea llamar hay que indicar entre paréntesis cuáles son los valores que se desea dar a los parámetros del método al hacer la llamada. O sea, se escribe:

<objeto>.<método>(<parámetros>)

Como es lógico, si el método no tomase parámetros se los dejaría vacío. Por ejemplo, para llamar al método Cumpleaños() de un objeto Persona llamado p se haría:

```
p.Cumpleaños();   // El método no toma parámetros, luego no le pasamos
ninguno
```

Es importante señalar que en una misma clase pueden definirse varios métodos con el mismo nombre siempre y cuando tomen diferente número o tipo de parámetros. A esto se le conoce como **sobrecargar** de métodos.

En C# todo, incluido los literales, son objetos del tipo de cada literal y por tanto pueden contar con miembros a los que se accedería tal y como se ha explicado. Para entender esto no hay nada mejor que un ejemplo:

```
string s = 12.ToString();
```

Este código almacena el literal de cadena "12" en la variable s, pues 12 es un objeto de tipo int (tipo que representa enteros) y cuenta con el método común a todos los ints llamado **ToString**() que lo que hace es devolver una cadena cuyos

caracteres son los dígitos.

Creación de objetos

Sintaxis:

new <nombreTipo>(<parametros>)

Este operador crea un nuevo objeto del tipo cuyo nombre se le indica y llama durante su proceso de creación al constructor del mismo apropiado según los valores que se le pasen en <parametros>, devolviendo una referencia al objeto recién creado.

El **constructor** recibe ese nombre debido a que su código suele usarse precisamente para construir el objeto, para inicializar sus miembros. Por ejemplo, a nuestra clase de ejemplo Persona le podríamos añadir un constructor dejándola así:

```
class Persona
{
  private string _nombre;  // Campo de cada objeto Persona que almacena su nombre
  private int _edad;       // Campo de cada objeto Persona que almacena su edad

// Constructor
public Persona (string nombre, int edad)
{
   _nombre = nombre;
   _edad = edad;
  }

   // Incrementa en uno la edad del objeto Persona
   void Cumpleaños()  {    _edad++;  }
}
```

Como se ve en el código, el constructor toma como parámetros los valores con los que deseemos inicializar el objeto a crear. Gracias a él, podemos crear un objeto Persona de nombre José, de 22 años de edad así:

new Persona("José", 22)

En realidad un objeto puede tener múltiples constructores, aunque para diferenciar a unos de otros es obligatorio que se diferencien en el número u orden de los parámetros que aceptan, ya que el nombre de todos ellos ha de coincidir con el nombre de la clase de la que son miembros.

Una vez creado un objeto lo más normal es almacenar la dirección devuelta por new en una variable del tipo apropiado para el objeto creado.

```
Persona p;                      // Creamos variable p
p = new Persona("Jose", 22);    // Almacenamos en p el objeto creado con new
```

Las instrucciones anteriores pueden compactarse en una sola así:

```
Persona p = new Persona("José", 22);
```

De hecho, una sintaxis más general para la definición de variables es la siguiente:

<tipoDato> <nombreVariable> = <valorInicial>;

La parte = <valorInicial> de esta sintaxis es en realidad opcional, y si no se incluye la variable declarada pasará a almacenar una referencia nula (contendrá el literal null)

Constructor por defecto

No es obligatorio definir un constructor para cada clase, y en caso de que no definamos ninguno el compilador creará uno por nosotros sin parámetros ni instrucciones. Es decir, como si se hubiese definido de esta forma:

<nombreTipo>(){}

Referencia al objeto actual con this

Dentro del código de cualquier método de un objeto siempre es posible hacer referencia al propio objeto usando la palabra reservada this. Esto puede venir bien a la hora de escribir constructores de objetos debido a que permite que los nombres que demos a los parámetros del constructor puedan coincidir nombres de los campos del objeto sin que haya ningún problema. Por ejemplo, el constructor de la clase Persona escrito anteriormente se puede reescribir así usando this:

```
Public Persona (string nombre, int edad) {
  this._nombre = nombre;
  this._edad = edad;
}
```

Es decir, dentro de un método con parámetros cuyos nombres coincidan con campos, se da preferencia a los parámetros y para hacer referencia a los campos hay que prefijarlos con el this tal y como se muestra en el ejemplo.

Un uso más frecuente de this en C# es el de permitir realizar llamadas a un método de un objeto desde código ubicado en métodos del mismo objeto. Es decir, en C# siempre es necesario que cuando llamemos a algún método de un objeto precedamos al operador . de alguna expresión que indique cuál es el objeto a cuyo método se desea llamar, y si éste método pertenece al mismo objeto que hace la llamada la única forma de conseguir indicarlo en C# es usando this.

Finalmente, una tercera utilidad de this es permitir escribir métodos que puedan devolver como objeto el propio objeto sobre el que el método es aplicado. Para ello bastaría usar una instrucción return this; al indicar el objeto a devolver.

Definición de variables

Para poder utilizar una variable hay que definirla indicando cual erá su nombre y cual será el tipo de datos que podrá almacenar, lo que se hace siguiendo la siguiente sintaxis:

<tipoVariable> <nombreVariable>;

Una variable puede ser definida dentro de una definición de clase, en cuyo caso se correspondería con el tipo de miembro que hasta ahora hemos denominado campo.

También puede definirse como un variable local a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código.

Otra posibilidad es definirla como parámetro de un método, que son variables que almacenan los valores de llamada al método y que, al igual que las variables locales, sólo puede ser accedida desde código ubicado dentro del método. El siguiente ejemplo muestra cómo definir variables de todos estos casos:

```
class A  {
      int x, z;
      int y;
      void F(string a, string b)    {     Persona p;    }
}
```

En este ejemplo las variables x, z e y son **campos** de tipo int, mientras que p es una variable **local** de tipo Persona y a y b son **parámetros** de tipo string.

Con la sintaxis de definición de variables anteriormente dada simplemente definimos variables pero no almacenamos ningún objeto inicial en ellas. El compilador dará un valor por defecto a los campos para los que no se indique explícitamente ningún valor. Pero no a la variables locales a las que no les da ningún valor inicial, pero detecta cualquier intento de leerlas antes de darles valor y produce errores de compilación en esos casos.

Por tanto, una forma de asignar un valor a la variable p del ejemplo anterior sería así:

```
Persona p;
p = new Persona("José", 22");
```

Una sintaxis más sencilla:

<tipoVariable> <nombreVariable> = <valorInicial>;

Así por ejemplo: Persona p = new Persona("José", 22);

También:

```
Persona p1 = new Persona("José", 2),  p2 = new Persona("Juan", 1);
```

Se recuerda que la Universidad de Alicante determina que los nombres de variables, campos y parámetros siga una notación Camel (ej: holaMundo, taxi)

Ejercicio 2:

Para resolver el problema se necesita trabajar con las fechas.

Para obtener el día de hoy `DateTime hoy = DateTime.Today;`

Para obtener el día de la semana de una fecha:

```
int diaSemana = (int) hoy.DayOfWeek;
string diaSemana = hoy.DayOfWeek.ToString();
```

Para sumar días a una fecha:

```
DatTime manana = hoy.AddDays(1);
```

Para restar fechas:

```
int dias =  hoy.Subtract(fecha).Days;
```

La resta de 2 fechas da como resultado un objeto TimeSpan, que tiene como propiedades días, horas, minutos y segundos.

A partir de 3 cajas de texto: nombre, fecha nacimiento y fecha inicio del curso y un botón de confirmación resuelve el siguiente problema:

1. Crea una clase que tenga 3 campos: Nombre, fecha nacimiento y fecha inicio de curso.

2. Si al cargar la página llega nombre y fecha de nacimiento entonces crearemos un objeto. Este tendrá como nombre y fecha nacimiento lo que le llegue. El campo inicio del curso tendrá como valor el día correspondiente al del próximo lunes (esto lo hará el constructor) (el cálculo lo realizará el método del punto 4). Ocurrirá lo mismo si el campo de fecha inicio del curso no es fecha válida.

3. Si llegan todos los datos, crearemos un objeto cuyos campos será lo que el usuario introdujo.

4. Esta clase tendrá un método que calculará la fecha del lunes siguiente (no es necesario hacer bucles o condiciones).

5. Esta clase tendrá un método que calculará la edad de la personas.

6. [Punto optativo] Esta clase tendrá un método que calculará los días que faltan para comenzar las clases (puede resultar un número negativo)

7. Finalmente, el constructor llamará a un método que mostrará un mensaje. El mensaje pondrá:

   "El alumno " NOMBRE " tiene "EDAD" años. El curso comienza el día "FECHA COMIENZO" dentro de "DIAS FALTAN" días.

```
    string texto = "El alumno " + nombre + " tiene " +
          CalculaEdad().ToString() + " años. El curso comienza el día "
          + fechaInicioCurso.ToString("dd/MM/yyyy") + " dentro de " +
          DiasFaltan().ToString() + " días ";
```

Nota: Observa esta forma de sobrecargar el constructor.

```
public Persona(string nombre, DateTime fechaNacimiento, DateTime
fechaInicioCurso) : this(nombre, fechaNacimiento)
    {
        this.fechaInicioCurso = fechaInicioCurso;
        Mensaje();
    }

    public Persona(string nombre, DateTime fechaNacimiento)
    {
        this.nombre = nombre;
        this.fechaNacimiento = fechaNacimiento;
        this.fechaInicioCurso = this.ProximoLunes();
        Mensaje();
    }
```

Propiedades

Las propiedades son miembros que ofrecen un mecanismo flexible para leer, escribir o calcular los valores de campos privados. Se pueden utilizar las propiedades como si fuesen miembros de datos públicos, aunque en realidad son métodos especiales denominados descriptores de acceso. De este modo, se puede tener acceso a los datos con facilidad, a la vez que proporciona la seguridad y flexibilidad de los métodos.

En este ejemplo, la clase TimePeriod almacena un período de tiempo. Internamente, la clase almacena el tiempo en segundos, pero se proporciona una propiedad denominada Hours que permite que un cliente especifique el tiempo en horas. Los descriptores de acceso de la propiedad Hours realizan la conversión entre horas y segundos.

```
class TimePeriod
{
    private double _seconds;
    public double hours
    {
        get { return _seconds / 3600; }
        set { _seconds = value * 3600; }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();
        // Asignando Hours a la propiedad.
```

```
        // Se llama al set de la propiedad
        t.Hours = 24;
        // Evaluando la propiedad Hours.
        // Se llama al get de la propiedad
        HttpContext.Current.Response.Write ("En horas: " + t.Hours);
    }
}
```

Se recuerda que la Universidad de Alicante determina que los nombres de propiedades siga una notación Camel (ej: holaMundo, taxi)

Ejercicio 3:

A partir de 2 cajas de texto: nombre y fecha nacimiento, un botón de confirmación y una etiqueta resultado resuelve el siguiente problema:

1. Crea una clase que tenga 3 campos: Nombre, fecha nacimiento y salida que será tipo Label.

2. Crea 4 propiedades:
    a. Propiedad nombre. Si al cargar la página llega nombre por post la propiedad asignará el valor al campo nombre. Valor por defecto cadena vacía.
    b. Propiedad fecha nacimiento.
    c. Propiedad edad. Si al cargar la página llega fecha nacimiento por post esta propiedad asignará la edad al campo edad. Valor por defecto -1.
    d. Propiedad label llamada salida. Será un control de la página aspx que recogerá el texto de salida.

3. Crea un método que escriba en un control label de la página aspx este texto: El alumno NOMBRE tiene EDAD años.

Arrays

Arrays unidimensionales

Es un tipo especial de variable que es capaz de almacenar en su interior y de manera ordenada uno o varios datos de un determinado tipo. Para declarar variables de este tipo especial se usa la siguiente sintaxis:

<tipoDatos>[] <nombreArray>;

Por ejemplo, un array que pueda almacenar objetos de tipo int se declara así:

int[] array;

Con esto la array creada no almacenaría ningún objeto, sino que valdría null. Si se desea que verdaderamente almacene objetos hay que indicar cuál es el número de objetos que podrá almacenar, lo que puede hacerse usando la siguiente sintaxis al declararla:

<tipoDatos>[] <nombreArray > = new <tipoDatos>[<númeroDatos>];

Por ejemplo, un array que pueda almacenar 100 objetos de tipo int se declara así:

int[] array = new int[100];

Aunque también sería posible definir el tamaño del array de forma separada a su declaración de este modo:

```
int[] array;
array = new int[100];
```

Con esta última sintaxis es posible cambiar dinámicamente el número de elementos de una variable array sin más que irle asignando nuevos array. Ello no significa que se pueda redimensionar conservando los elementos que tuviese antes del cambio de tamaño, sino que ocurre todo lo contrario: cuando a una variable array se le asigna un array de otro tamaño, sus elementos antiguos son sobreescritos por los nuevos.

Si queremos darles otros valores diferentes a los que tomaría por defecto según el tipo de dato al declarar el array:

<tipoDatos>[] <nombreTabla> = new <tipoDatos>[] {<valores>};

Poniendo tantos <valores> como número de elementos se desee que tenga el array separados por comas (,):

int[] tabla = new int[] {5,1,4,0};

Incluso se puede compactar aún más la sintaxis declarando la tabla así:

int[] tabla = {5,1,4,0};

A la hora de acceder a los elementos almacenados en una tabla basta indicar entre corchetes, y a continuación de la referencia a la misma, la posición que ocupe en el array el elemento al que acceder. Hay que tener en cuenta que en C# los arrays se indexan desde 0.

Si se especifica una posición superior al número de elementos que pueda almacenar se producirá una excepción de tipo System.OutOfBoundsException. Para evitar este tipo de excepciones puede consultar el valor del campo de sólo lectura Length que

está asociado a todo array y contiene el número de elementos de la misma. Por ejemplo, para asignar un 7 al último elemento de la:

```
array[array.Length - 1] = 7;  // Se resta 1 pq aunque devuelve 4 el último es array[3]
```

Un array puede contener elementos que a su vez sean arrays, llamándose **arrays dentados**.

```
int[][] ArrayDentado = new int[2][] {new int[] {1,2}, new int[] {3,4,5}};
```

```
int[][] ArrayDentado = {new int[] {1,2}, new int[] {3,4,5}};
```

Es importante señalar que no es posible especificar todas las dimensiones de un array dentado en su definición si no se indica explícitamente el valor inicial de éstas entre llaves. Es decir, esta declaración es **incorrecta**:

```
int[][] ArrayDentado = new int[2][5];
```

Así si sería correcto:

```
int[][] ArrayDentado = {new int[5], new int[5]};
```

Arrays multidimensionales

Para definir este tipo de tablas se usa una sintaxis similar a la usada para declarar tablas unidimensionales pero separando las diferentes dimensiones mediante comas (,) Por ejemplo, uno de elementos de tipo int que conste de 12 elementos puede tener sus elementos distribuidos en dos dimensiones formando una estructura 3x4 similar a una matriz de la forma:

```
1  2  3  4
5  6  7  8
9 10 11 12
```

Esta tabla se podría declarar así:

```
int[,] arrayMultidimensional = new int[3,4]
```

Parámetros

Valores en Parámetros in

Se utilizan valores en parámetros para pasar una variable por valor a un método, la variable del método es inicializada con una copia del valor del caller (quien realizó la invocación). Si no hay modificadores los parámetros son siempre pasados por valor.

Valores en Parámetros ref

Es posible pasar un valor como parámetro a un método, modificar el valor y regresarlo como resultado del método, para ello se utiliza el modificador ref seguido del tipo y del nombre del parámetro.

Al contrario de los valores en parámetros no se pasa una copia del valor, sino la referencia del valor y por ello al modificar el valor se hace la modificación directa, también es necesario inicializar el valor que se pasa como paramétro por medio de una variable intermedia y no directamente a través de una expresión constante:

```
using System;

public class Param
{
        public void ParametroRef(ref  int  RefParametro)
        {                                                   //No regresa un valor explícito

                RefParametro *= RefParametro;     //Se modifica el valor directamente
                                                  //No se regresa un valor
                                                  /porque se modifico de manera directa
        }
}
```

```
public partial class Default8 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
      {
                Param MiParam = new Param();

                int   iValorRef = 5;                              //Se requiere inicializar el valor
                MiParam.ParametroRef(ref  iValorRef);
                                        //Se invoca el método  pasando la referencia del valor
                Response.Write("ref : " + iValorRef);
        }
}
```

Resultado: ref: 25

Good Practice, se recomienda tener dos variables, una en el parámetro y otra en el parámetro ref.

Valores en Parámetros out

Un parámetro out puede ser utilizado sólo para contener el resultado de un método, es necesario especificar el modificador out para indicar el tipo de parámetro, a diferencia de los parámetros ref el caller no necesita inicializar la variable antes de invocar el método:

```
using System;
public class Param
{
        public void ParametroOut(out int OutParametro)
        {
                OutParametro = 4 * 4;       //No devuelve un valor, porque es dev en el parámetro out
        }
}

public partial class Default8 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
       {
                Param MiParam = new Param();
                int   iValorOut;                          //No se requiere inicializar el valor
                MiParam.ParametroOut(out  iValorOut);     //Se invoca el método con un parámetro out
                Response.Write("out : " + iValorOut);     //Resultado de la invocación del método
        }
}
```

Resultado = out: 16

Instrucciones

Instrucciones condicionales

Instrucción if

La instrucción if permite ejecutar ciertas instrucciones sólo si se da una determinada condición. Su sintaxis de uso es la sintaxis:

```
if (<condición>)
  <instruccionesIf>
else
  <instruccionesElse>
```

Instrucción switch

La instrucción switch permite ejecutar unos u otros bloques de instrucciones según el valor de una cierta expresión. Su estructura es:

```
switch (<expresión>) {
  case <valor1>:  <bloque1>
                                <siguienteAcción>
  case <valor2>:  <bloque2>
                                <siguienteAcción>
                    ...
  default:  <bloqueDefault>
                            <siguienteAcción>
}
```

El significado de esta instrucción es el siguiente: se evalúa <expresión>. Si su valor es <valor1> se ejecuta el <bloque1>, si es <valor2> se ejecuta <bloque2>, y así para el resto de valores especificados. Si no es igual a ninguno de esos valores y se incluye la rama default, se ejecuta el <bloqueDefault>; pero si no se incluye se pasa directamente a ejecutar la instrucción siguiente al switch .

En realidad, aunque todas las ramas de un switch son opcionales siempre se ha de incluir al menos una. Además, la rama default no tiene porqué aparecer la última si se usa, aunque es recomendable que lo haga para facilitar la legibilidad del código.

El elemento marcado como <siguienteAcción> colocado tras cada bloque de instrucciones indica qué es lo que ha de hacerse tras ejecutar las instrucciones del bloque que lo preceden. Puede ser uno de estos tres tipos de instrucciones:

goto case <valori>;
goto default;
break;

Si es un goto case indica que se ha de seguir ejecutando el bloque de instrucciones asociado en el switch a la rama del <valori> indicado, si es un goto default indica que se ha de seguir ejecutando el bloque de instrucciones de la rama default, y si es un break indica que se ha de seguir ejecutando la instrucción siguiente al switch.

Instrucciones iterativas

Instrucción while

La instrucción while permite ejecutar un bloque de instrucciones mientras se dé una cierta instrucción. Su sintaxis de uso es:

while (<condición>) <instrucciones>

Su significado es el siguiente: Se evalúa la <condición> indicada, que ha de producir un valor lógico. Si es cierta (valor lógico true) se ejecutan las <instrucciones> y se repite el proceso de evaluación de <condición> y ejecución de <instrucciones> hasta que deje de serlo. Cuando sea falsa (false) se pasará a ejecutar la instrucción siguiente al while.

Por otro lado, dentro de las <instrucciones> de un while pueden usarse dos instrucciones especiales:

- break;: Indica que se ha de abortar la ejecución del bucle y continuarse ejecutando por la instrucción siguiente al while.

- continue;: Indica que se ha de abortar la ejecución de las <instrucciones> y reevaluarse la <condición> del bucle, volviéndose a ejecutar la <instrucciones> si es cierta o pasándose a ejecutar la instrucción siguiente al while si es falsa.

Instrucción do...while

La instrucción do...while es una variante del while que se usa así:

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

La única diferencia del significado de do...while respecto al de while es que en vez de evaluar primero la condición y ejecutar <instrucciones> sólo si es cierta, **do...while primero ejecuta las <instrucciones> y luego mira la <condición>** para ver si se ha de repetir la ejecución de las mismas. Por lo demás ambas instrucciones son iguales, e incluso también puede incluirse break; y continue; entre las <instrucciones> del do...while.

do ... while está especialmente destinado para los casos en los que haya que ejecutar las <instrucciones> **al menos una vez** aún cuando la condición sea falsa desde el principio., como ocurre en el siguiente ejemplo:

```
using System;

 class HolaMundoDoWhile {
   public static void Main()   {
    String leído;
        do {
           Console.WriteLine("Clave: ");
            leído = Console.ReadLine();
        }    while (leído != "José");
         Console.WriteLine("Hola José");
    }
}
```

Este programa pregunta al usuario una clave y mientras no introduzca la correcta (José) no continuará ejecutándose. Una vez que introducida correctamente dará un mensaje de bienvenida al usuario.

Instrucción for

La instrucción for es una variante de while que permite reducir el código necesario para escribir los tipos de bucles más comúnmente usados en programación. Su sintaxis es:

```
for (<inicialización>; <condición>; <modificación>)
  <instrucciones>
```

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de <inicialización>, que suelen usarse para definir e inicializar variables que luego se usarán en <instrucciones>. Luego se evalúa <condición>, y si es falsa se continúa ejecutando por la instrucción siguiente al for; mientras que si es cierta se ejecutan las <instrucciones> indicadas, luego se ejecutan las instrucciones de <modificación> -que como su nombre indica suelen usarse para modificar los valores de variables que se

usen en <instrucciones>- y luego se reevalúa <condición> repitiéndose el proceso hasta que ésta última deje de ser cierta.

En <inicialización> puede en realidad incluirse cualquier número de instrucciones que no tienen porqué ser relativas a inicializar variables o modificarlas, aunque lo anteriro sea su uso más habitual. En caso de ser varias se han de separar mediante comas (,), ya que el carácter de punto y coma (;) habitualmente usado para estos menesteres se usa en el for para separar los bloques de <inicialización>, <condición> y <modificación>. Además, la instrucción nula no se puede usar en este caso y tampoco pueden combinarse definiciones de variables con instrucciones de otros tipos.

Con <modificación> pasa algo similar, ya que puede incluirse código que nada tenga que ver con modificaciones pero en este caso no se pueden incluir definiciones de variables.

Las variables que se definan en <inicialización> serán visibles sólo dentro de esas <instrucciones>

Instrucción foreach

La instrucción foreach es una variante del for pensada especialmente para compactar la escritura de códigos donde se realice algún tratamiento a todos los elementos de una colección, que suele un uso muy habitual de for en los lenguajes de programación que lo incluyen. La sintaxis que se sigue a la hora de escribir esta instrucción foreach es:

```
foreach (<tipoElemento> <elemento> in <colección>)
  <instrucciones>
```

El significado de esta instrucción es muy sencillo: se ejecutan <instrucciones> para cada uno de los elementos de la <colección> indicada. <elemento> es una variable de sólo lectura de tipo <tipoElemento> que almacenará en cada momento el elemento de la colección que se esté procesando y que podrá ser accedida desde <instrucciones>.

Es importante señalar que <colección> no puede valer null porque entonces saltaría una excepción de tipo System.NullReferenceException, y que <tipoElemento> ha de ser un tipo cuyos objetos puedan almacenar los valores de los elementos de <colección>

Direcciones de interés

http://www.programacion.com/tutorial/csharp/

http://www.elguille.info/Net/dotnet/equivalenciavbcs1.htm

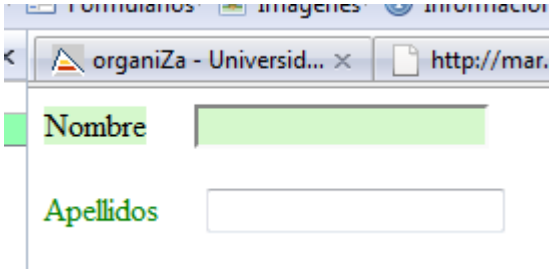
http://es.csharp-online.net/C_sharp_NET_/_Cap%C3%ADtulo_4

http://www.es-asp.net/tutoriales-asp-net/tutorial-61-109/contenidos-de-un-tema-y-skin.aspx

Ejercicio 4:

Dado un formulario con 4 campos:

```
<asp:Label ID="nombre" runat="server" Text="Nombre" SkinID="obl">
</asp:Label>
<asp:TextBox ID="TextBox1" runat="server" SkinID="obl">
</asp:TextBox>
 <br /><br />
<asp:Label ID="apellidos" runat="server" Text="Apellidos">
</asp:Label>
<asp:TextBox ID="TextBox2" runat="server">
</asp:TextBox>
```



1. Saca por pantalla la relación de los controles que pones en el formulario.

```
Page.FindControl("nombreFormulario").Controls
```

2. De los controles que salen selecciona los que son “Label” con SkinId = Obl y concatena al final un “*”. Según el ejemplo sería “Nombre *”.

3. De los controles que salen añade un “RequiredFieldValidator” a los textbox marcados con SkinId = Obl.

```
RequiredFieldValidator RV1 = new RequiredFieldValidator();
RV1.ID = "RV1";
RV1.ControlToValidate = "TxtNombre";
RV1.ErrorMessage = "Error... introduzca un nombre";
micontrol2.Parent.Controls.Add(RV1);
```

ANEXO 1

Herencia y métodos virtuales

Mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera -a la que se le suele llamar clase hija- será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda -a la que se le suele llamar clase padre o clase base. Las clases que derivan de otras se definen usando la siguiente sintaxis:

class <nombreHija>:<nombrePadre> { <miembrosHija> }

A los miembros definidos en <miembrosHijas> se le añadirán los que hubiésemos definido en la clase padre. Por ejemplo, a partir de la clase Persona puede crearse una clase Trabajador así:

```
class Trabajador: Persona  {
  public int Sueldo;

    public Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad)
    {
        Sueldo = sueldo;
    }
}
```

Los objetos de esta clase Trabajador contarán con los mismos miembros que los objetos Persona y además incorporarán un nuevo campo llamado Sueldo que almacenará el dinero que cada trabajador gane.

A la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial consistente en preceder la llave de apertura del cuerpo del método de una estructura de la forma:

: base(<parametrosBase>)

A esta estructura se le llama inicializador base y se utiliza para indicar cómo deseamos inicializar los campos heredados de la clase padre. No es más que una llamada al constructor de la misma con los parámetros adecuados, y si no se incluye el compilador consideraría por defecto que vale :base(), lo que sería incorrecto en este ejemplo debido a que Persona carece de constructor sin parámetros.

Un ejemplo que pone de manifiesto cómo funciona la herencia es el siguiente:

```
using System;

class Persona {
  public string Nombre; // Campo de cada objeto Persona que almacena su nombre
  public int Edad;      // Campo de cada objeto Persona que almacena su edad

  void Cumpleaños()  {   Edad++;   }
```

```
  // Constructor de Persona
  public Persona (string nombre, int edad)    {
    Nombre = nombre;
    Edad = edad;
  }
}

class Trabajador: Persona {
  public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  // Inicializamos cada Trabajador en base al constructor de Persona
  Trabajador(string nombre, int edad, string nif, int sueldo):  base(nombre, edad)   {
        Sueldo = sueldo;
  }
}
```

Ha sido necesario prefijar la definición de los miembros de Persona del palabra reservada public: por defecto los miembros de un tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo public conseguimos que sean accesibles desde cualquier código.

public es lo que se denomina un modificador de acceso

Llamadas por defecto al constructor base

Si en la definición del constructor de alguna clase que derive de otra no incluimos inicializador base el compilador considerará que éste es :base() Por ello hay que estar seguros de que si no se incluye base en la definición de algún constructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

Métodos virtuales

Es posible cambiar la definición en la clase hija, para lo que habría que haber precedido con la palabra reservada virtual la definición de dicho método en la clase padre. A este tipo de métodos se les llama métodos virtuales, y la sintaxis que se usa para definirlos es la siguiente:

virtual <tipoDevuelto> <nombreMétodo>(<parámetros>){ <código> }

Si en alguna clase hija quisiésemos dar una nueva definición del <código> del método, simplemente lo volveríamos a definir en la misma pero sustituyendo en su definición la palabra reservada virtual por override. Es decir, usaríamos esta sintaxis:

override <tipoDevuelto> <nombreMétodo>(<parámetros>){ <nuevoCódigo> }

Nótese que esta posibilidad de cambiar el código de un método en su clase hija sólo se da si en **la clase padre el método fue definido como virtual**. En caso contrario, el compilador considerará un error intentar redefinirlo.

Para aclarar mejor el concepto de método virtual, vamos a mostrar un ejemplo en el que cambiaremos la definición del método Cumpleaños() en los objetos Persona por una nueva versión en la que se muestre un mensaje cada vez que se ejecute, y redefiniremos dicha nueva versión para los objetos Trabajador de modo que el mensaje mostrado sea otro. El código de este ejemplo es el que se muestra a continuación:

```
using System;

class Persona {
  public string Nombre;      // Campo de cada objeto Persona que almacena su nombre
  public int Edad;           // Campo de cada objeto Persona que almacena su edad
  public string NIF;         // Campo de cada objeto Persona que almacena su NIF

  // Incrementa en uno de la edad del objeto Persona
  public virtual void Cumpleaños()  {   Console.WriteLine("Incrementada edad de  persona");  }

  // Constructor de Persona
  public Persona (string nombre, int edad, string nif)  {
    Nombre = nombre;
    Edad = edad;
    NIF = nif;
  }
}

class Trabajador: Persona {
  public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  // Inicializamos cada Trabajador en base al constructor de Persona
  Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)  {
              Sueldo = sueldo;
  }

  public override Cumpleaños()  {
    Edad++;
    Console.WriteLine("Incrementada edad de persona");
  }

  public static void Main()  {
     Persona p = new Persona("Carlos", 22, "77588261-Z", 100000);
    Trabajador t = new Trabajador("Josan", 22, "77588260-Z", 100000);

    t.Cumpleaños();
    p.Cumpleaños();
  }
}
```

Nótese cómo se ha añadido el modificador virtual en la definición de Cumpleaños() en la clase Persona para habilitar la posibilidad de que dicho método puede ser redefinido en clase hijas de Persona y cómo se ha añado override en la redefinición del mismo dentro de la clase Trabajador para indicar que la nueva definición del método es una redefinición del heredado de la clase. La salida de este programa confirma que la implementación de Cumpleaños() es distinta en cada clase, pues es de la forma:

```
Incrementada edad de trabajador
Incrementada edad de persona
```

También es importante señalar que para que la redefinición sea válida ha sido necesario añadir public a la definición del método original, pues si no se consideraría que el método sólo es accesible desde dentro de la clase donde se ha definido, lo que no tiene sentido en métodos virtuales ya que entonces nunca podría ser redefinido.

Además, este modificador también se ha mantenido en la redefinición de Cumpleaños() porque toda redefinición de un método virtual ha de mantener los mismos modificadores de acceso que el método original para ser válida.

Clases abstractas

Una clase abstracta es aquella que forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma o acceder a sus miembros estáticos Para definir una clase abstracta se antepone abstract a su definición, como se muestra en el siguiente ejemplo:

```
public abstract class A{        public abstract void F();      }
abstract public class B: A{     public void G() {}             }
class C: B{                     public override void F()  {}   }
```

Las clases A y B del ejemplo son abstractas, y como puede verse es posible combinar en cualquier orden el modificador abstract con modificadores de acceso.

La utilidad de las clases abstractas es que pueden contener métodos para los que no se dé directamente una implementación sino que se deje en manos de sus clases hijas darla. No es obligatorio que las clases abstractas contengan métodos de este tipo, pero sí lo es marcar como abstracta a toda la que tenga alguno. Estos métodos se definen precediendo su definición del modificador abstract y sustituyendo su código por un punto y coma (;), como se muestra en el método F() de la clase A del ejemplo (nótese que B también ha de definirse como abstracta porque tampoco implementa el método F() que hereda de A)