**GUIA DE PROGRAMACIÓN C#**

**Origen de C#**

**C#** (leído en inglés "C Sharp") es el nuevo lenguaje de propósito

general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott

Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del

lenguaje Turbo Pascal y la herramienta RAD Delphi.

Aunque es posible escribir código para la plataforma .NET en muchos otros lenguajes, C# es el

único que ha sido diseñado específicamente para ser utilizado en ella, por lo que programarla

usando C# es mucho más sencillo e intuitivo que hacerlo con cualquiera de los otros lenguajes ya

que C# carece de elementos heredados innecesarios en .NET. Por esta razón, se suele decir que C# es el **lenguaje nativo de .NET**

La sintaxis y estructuración de C# es muy similar a la C++, ya que la intención de Microsoft con C# es facilitar la migración de códigos escritos en estos lenguajes a C# y facilitar su aprendizaje a los desarrolladores habituados a ellos. Sin embargo, su sencillez y el alto nivel de productividad son equiparables a los de Visual Basic.

**Características de C#**

**Sencillez:** C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET. Por ejemplo:

* El código escrito en C# es **autocontenido**, lo que significa que no necesita

de ficheros adicionales al propio fuente tales como ficheros de cabecera o

ficheros IDL.

* El tamaño de los tipos de datos básicos es fijo e independiente del

compilador, sistema operativo o máquina para quienes se compile (no como

en C++), lo que facilita la portabilidad del código.

* No se incluyen elementos poco útiles de lenguajes como C++ tales como

macros, herencia múltiple o la necesidad de un operador diferente del punto

(**.**) acceder a miembros de espacios de nombres (**::**)

**Modernidad:** C# incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como Java o C++ hay que simular, como un tipo básico **decimal** que permita realizar operaciones de alta precisión con reales de 128 bits (muy útil en el mundo financiero), la inclusión de una instrucción **foreach** que permita recorrer colecciones con facilidad y es ampliable a tipos definidos por el

usuario, la inclusión de un tipo básico **string** para representar cadenas o la distinción de un tipo **bool** específico para representar valores lógicos.

**Orientación a objetos:** Como todo lenguaje de programación de propósito general actual, C# es un lenguaje orientado a objetos, aunque eso es más bien una característica del CTS que de C#. Una diferencia de este enfoque orientado a objetos respecto al de otros lenguajes como C++ es que el de C# es más puro en tanto que no admiten ni funciones ni variables globales sino que todo el código y

datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.

C# soporta todas las características propias del paradigma de programación orientada a

objetos: **encapsulación**, **herencia** y **polimorfismo**.

**Escritura de Aplicaciones**

Básicamente una aplicación en C# puede verse como un conjunto de uno o más ficheros de código

fuente con las instrucciones necesarias para que la aplicación funcione como se desea y que son

pasados al compilador para que genere un ejecutable. Cada uno de estos ficheros no es más que un

fichero de texto plano escrito usando caracteres Unicode y siguiendo la sintaxis propia de C#.

Como primer contacto con el lenguaje, nada mejor que el típico programa de iniciación "¡Hola

Mundo!" que lo único que hace al ejecutarse es mostrar por pantalla el mensaje ¡Hola Mundo! Su

código es:

1: class HolaMundo

2: {

3: static void Main()

4: {

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

6: }

7: }

Todo el código escrito en C# se ha de escribir dentro de una definición de clase, y lo que en la línea **1:** se dice es que se va a definir una clase (**class**) de nombre HolaMundo1 cuya definición estará comprendida entre la llave de apertura de la línea **2:** y su correspondiente llave de cierre en la línea

línea **7:** Dentro de la definición de la clase (línea **3:**) se define un método de nombre Main cuyo código es elindicado entre la llave de apertura de la línea **4:** y su respectiva llave de cierre (línea **6:**) Un métodono es más que un conjunto de instrucciones a las que se les asocia un nombre, de modo que paraposteriormente ejecutarlas baste 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 método, y en este caso es **void** que significa que no se devuelve nada. Por su parte, los paréntesis que se colocado tras el nombre del método indican cuáles son los parámetros éste toma, y como en este caso están vacíos ello significa que el método no toma parámetros. Los

parámetros de un método permiten variar el resultado de su ejecución según los valores que se les dé en cada llamada.

La palabra **static** que antecede a la declaración del tipo de valor devuelto es un **modificador** del significado de la declaración de método que indica que el método está asociado a la clase dentro de

la que se define y no a los objetos que se creen a partir de ella. Main() es lo que es denomina el **punto de entrada** de la aplicación, que no es más que el método por el que comenzará su ejecución. Necesita del modificador **static** para evitar que para llamarlo haya que crear algún objeto de la clase donde se haya definido.

Finalmente, la línea **5:** contiene la instrucción con el código a ejecutar, que lo que se hace es solicitar la ejecución del método **WriteLine()** de la clase **Console** definida en el espacio de nombres **System** pasándole como parámetro la cadena de texto con el contenido ¡Hola Mundo!

Nótese que las cadenas de textos son secuencias de caracteres delimitadas por comillas dobles aunque dichas comillas no forman parte de la cadena. Por su parte, un espacio de nombres puede considerarse que es algo similar para las clases a lo que un directorio es para los ficheros; es decir,

es una forma de agruparlas.

**Punto de Entrada**

Ya se ha dicho que el **punto de entrada** de una aplicación es un método de nombre Main que

contendrá el código por donde se ha de iniciar la ejecución de la misma. Hasta ahora sólo se ha

visto una versión de Main() que no toma parámetros y tiene como tipo de retorno **void**, pero en

realidad todas sus posibles versiones son:

**static void Main()**

**static int Main()**

**static int Main(string[]** args**)**

**static void Main(string[]** args**)**

Como se ve, hay versiones de Main() que devuelven un valor de tipo **int**. Un **int** no es más que un

tipo de datos capaz de almacenar valor enteros comprendidos entre -2.1471483.648 y

2.1471483.647, y el número devuelto por Main() sería interpretado como código de retorno de la

aplicación. Éste valor suele usarse para indicar si la aplicación a terminado con éxito (generalmente

valor 0) o no (valor según la causa de la terminación anormal), y en el *Tema 8: Métodos* se

explicará como devolver valores.

También hay versiones de Main() que toman un parámetro donde se almacenará la lista de

argumentos con los que se llamó a la aplicación, por lo que sólo es útil usar estas versiones del

punto de entrada si la aplicación va a utilizar dichos argumentos para algo. El tipo de este

parámetro es **string[]**, lo que significa que es una tabla de cadenas de texto

y su nombre -que es el que habrá de usarse dentro del código de Main() para hacerle referencia- es args en el ejemplo, aunque podría dársele cualquier otro.

**Comentarios**

Un **comentario** es texto que incluido en el código fuente de un programa con la idea de facilitar su legibilidad a los programadores y cuyo contenido es, por defecto, completamente ignorado por el compilador. Suelen usarse para incluir información sobre el autor del código, para aclarar el significado o el porqué de determinadas secciones de código, para describir el funcionamiento de los métodos de las clases, etc.

En C# hay dos formas de escribir comentarios. La primera consiste en encerrar todo el texto que se desee comentar entre

caracteres /\* y \*/ siguiendo la siguiente sintaxis:

/\*<texto>\*/

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>

Y un ejemplo de su uso es:

// Este comentario ejemplifica como escribir comentarios abreviados de una sola línea

**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

**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.

A continuación se describen cuáles son los operadores incluidos en el lenguaje clasificados según el tipo de operaciones que permiten realizar, aunque hay que tener en cuenta que C# permite la redefinición del significado de la mayoría de los operadores según el tipo de dato sobre el que se apliquen, por lo que lo que aquí se cuenta se corresponde con los usos más comunes de los mismos:

l **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" (+).

Relacionados con las operaciones aritméticas se encuentran un par de operadores llamados **checked** y **unchecked** que permiten controlar si se desea detectar los desbordamientos que puedan producirse si al realizar este tipo de operaciones el resultado es superior a la capacidad del tipo de datos de sus operandos. Estos operadores se usan así:

**checked (**<expresiónAritmética>**)**

**unchecked(**<expresiónAritmética>**)**

Ambos operadores calculan el resultado de <expresiónAritmética> y lo devuelven si durante el cálculo no se produce ningún desbordamiento. Sin embargo, en caso de que haya desbordamiento cada uno actúa de una forma distinta: **checked** provoca un error de compilación si <expresiónAritmética> es una expresión constante y una excepción **System.OverflowException** si no lo es, mientras que **unchecked** devuelve el resultado de la expresión aritmética truncado para modo que quepa en el tamaño esperado.

Por defecto, en ausencia de los operadores **checked** y **unchecked** lo que se hace es evaluar las operaciones aritméticas entre datos constantes como si se les aplicase **checked** y las operaciones entre datos no constantes como si se les hubiese aplicado **unchecked**.

l **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.

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

l **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.

l **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, pues el operador compuesto +=lo que hace es asignar a su primer operando el valor que tenía más el valor de su segundo operando. Como se ve, permite compactar bastante el código.

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 usar asignación compuesta ni incremento

temperatura += 1; // Usando asignación compuesta

temperatura++; // Usando incremento

Si el operador ++ se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el

operador --. Por ejemplo:

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

La ventaja de usar los operadores ++ y -- es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador traducirlos en una única instrucción en código máquina.

l **Operaciones con cadenas:** Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos. Por ejemplo, "Hola"+" mundo" devuelve "Hola mundo", y "Hola mund" + "o" también.

l **Operaciones de acceso a tablas:** Una **tabla** es un conjunto de ordenado de objetos de tamaño fijo.

Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo [] sobre la tabla para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder dentro del conjunto. Es decir, este operador se usa así:

**[<**posiciónElemento>**]**

Un ejemplo de su uso en el que se asigna al elemento que ocupa la posición 3 en una tabla de nombre tablaPrueba el valor del elemento que ocupa la posición 18 de dicha tabla es el siguiente:

tablaPrueba[3] = tablaPrueba[18];

Las tablas se estudian detenidamente en el *Tema 7: Variables y tipos de datos.*

l **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>. Un ejemplo de su uso es:

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** que se tratará en el *Tema 8:Instrucciones*, pues

aunque su utilidad es similar al de ésta, ? devuelve un valor e if no.

l **Operaciones de delegados:** Un **delegado** es un objeto que puede almacenar en referencias a uno o más métodos y a través del cual es posible llamar a estos métodos. Para añadir objetos a un delegado se usan los operadores + y +=, mientras que para quitárselos se usan los operadores - y -=.

l **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

No se preocupe si no conoce los conceptos de métodos, propiedades, eventos y delegados en los que se basa este ejemplo, pues se explican detalladamente en temas posteriores.

l **Operaciones con punteros:** Un puntero es una variable que almacena una referencia a una dirección de memoria. Para obtener la dirección de memoria de un objeto se usa el operador &, para acceder al contenido de la dirección de memoria almacenada en un puntero se usa el operador \*, para acceder a un

miembro de un objeto cuya dirección se almacena en un puntero se usa ->, y para referenciar una dirección de memoria de forma relativa a un puntero se le aplica el operador [] de la forma puntero[desplazamiento].

l **Operaciones de obtención de información sobre tipos:** De todos los operadores que nos permiten obtener información sobre tipos de datos el más importante es typeof, cuya forma de uso es:

**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>

El significado de este operador es el siguiente: se evalúa <expresión>. Si el resultado de ésta es del tipo cuyo nombre se indica en <nombreTipo>se devuelve **true**; y si no, se devuelve **false**.

Finalmente, C# incorpora un tercer 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>**)**

**sizeof** sólo puede usarse dentro de código inseguro, que por ahora basta considerar que son zonas de código dondees posible usar punteros.

Además, **sizeof** sólo se puede aplicar sobre nombres de tipos de datos cuyos objetos se puedan almacenar directamente en pila.

l **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. En función del tipo y número de estos parámetros se llamará a uno u otro de los constructores del objeto. 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

En caso de que el tipo del que se haya solicitado la creación del objeto sea una clase, éste se creará en memoria dinámica, y lo que **new** devolverá será una referencia a la dirección de pila donde se almacena una referencia a la dirección del objeto en memoria dinámica. Sin embargo, si el objeto a crear pertenece a una estructura o a un tipo enumerado, entonces éste se creará directamente en la pila y la referencia devuelta por el **new** se referirá

directamente al objeto creado. Por estas razones, a las clases se les conoce como **tipos referencia** ya que de sus objetos en pila sólo se almacena una referencia a la dirección de memoria dinámica donde verdaderamente se encuentran; mientras que a las estructuras y tipos enumerados se les conoce como **tipos valor** ya sus objetos se almacenan directamente en pila.

C# proporciona otro operador que también nos permite crear objetos. Éste es **stackalloc**, y se usa así: **stackalloc** <nombreTipo>**[**<nElementos>**]**

Este operador lo que hace es crear en pila una tabla de tantos elementos de tipo <nombreTipo> como indique <nElementos> y devolver la dirección de memoria en que ésta ha sido creada. Por ejemplo: **stackalloc** sólo puede usarse para inicializar punteros a objetos de tipos valor declarados como variables locales. Por ejemplo:

int \* p = stackalloc[100]; // p apunta a una tabla de 100 enteros.

**Operaciones de conversión:** 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.

Existe otro operador que permite realizar operaciones de conversión de forma muy similar al ya visto. Éste es el operador **as**, que se usa así:

<expresión> **as** <tipoDestino>

Lo que hace es devolver el resultado de convertir el resultado de evaluar <expresión> al tipo indicado en <tipoDestino> Por ejemplo, para almacenar en una variable p el resultado de convertir un objeto t a tipo tipo Persona se haría:

p = t as Persona;

Las únicas diferencias entre usar uno u otro operador de conversión son:

m **as** sólo es aplicable a tipos referencia y sólo a aquellos casos en que existan conversiones predefinidas en el lenguaje. Como se verá más adelante, esto sólo incluye conversiones entre un tipo y tipos padres suyos y entre un tipo y tipos hijos suyos.

Una consecuencia de esto es que el programador puede definir cómo hacer conversiones de tipos por él definidos y otros mediante el operador **()**, pero no mediante **as**.

Esto se debe a que as únicamente indica que se desea que una referencia a un objeto en memoria dinámica se trate como si el objeto fuese de otro tipo, pero no implica conversión ninguna. Sin embargo, **()** sí que implica conversión si el <tipoDestino> no es compatible con el tipo del objeto referenciado. Obviamente, el operador se aplirará mucho más rápido en los casos donde no sea necesario convertir.

m En caso de que se solicite hacer una conversión inválida **as** devuelve **null** mientras que **()** produce una excepción **System.InvalidCastException**.

**Clases**

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**

La sintaxis básica para definir una clase es la que a continuación se muestra:

**class** <nombreClase>

**{**

*<miembros>*

**}**

De este modo se definiría una clase de nombre <nombreClase> cuyos miembros son los definidos en <miembros> Los **miembros** de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma. Un ejemplo de cómo declarar una clase de nombre A que no tenga ningún miembro es la siguiente:

class A{}

Una clase así declarada no dispondrá de ningún miembro a excepción de los implícitamente definidos de manera común para todos los objetos que creemos en C#.

El nombre de la clase deberá indicar la acción que realiza y una referencia al caso de uso a que pertenece como se muestra a continuación:

Class <Menu\_Usuarios>

{

<miembros>

}

/\* Primeramente llevará un nombre de la acción que realiza seguido por un nombre que identifique un caso de uso que igual podría ser el nombre del caso de uso, de la segunda palabra en adelante llevará el primer carácter mayúscula.

Class <consultaMantBit>{

<miembros>

}\*/

**Metodos**

**/tendremos dos tipos de nombramiento de métodos.**

**El primero es para los eventos que se estén incluidos en la clase los cuales deberán tener al principio el nombre del elemento y separado por un guion bajo la acción que activo el evento.**

**El segundo son los métodos que no llaman eventos y cuyo nombres deberá ser la acción que realiza y a continuación el tipo de datos o función que cumple el método/**

**Ejemplo 1:**

protected void btest\_Click(parametros)

**Ejemplo 2:**

private void cargar(parametros)

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, por lo que como al principio del tema se indicó, 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>*

**}**

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>*.

Tendremos dos tipos de nombramiento de métodos.

El primero es para los eventos que se estén incluidos en la clase los cuales deberán tener al principio el nombre del elemento y separado por un guion bajo la acción que activo el evento.

protected void btBuscar\_Click(object sender, EventArgs e)

{

}

El segundo son los métodos que no llaman eventos y cuyo nombres deberá ser la acción que realiza y a continuación el tipo de datos o función que cumple el método

private void cargarLista()

{

}

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

{

string Nombre; // Campo de cada objeto Persona que almacena su nombre

int Edad; // Campo de cada objeto Persona que almacena su edad

string NIF; // Campo de cada objeto Persona que almacena su NIF

void Cumpleaños() // Incrementa en uno de la edad del objeto Persona

{

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 dejarían vacíos los parámetros en la llamada al mismo. 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**, y es posible ya que cuando se les llame el compilador sabrá a cual llamar a partir de *<parámetros>* pasados en la llamada.

Sin embargo, lo que no es permite es definir varios métodos que sólo se diferencien en su valor de retorno, ya que como éste no se tiene porqué indicar al llamarlos no podría diferenciarse a que método en concreto se hace referencia en cada llamada. Por ejemplo, a partir de la llamada: p.Cumpleaños();

Si además de la versión de Cumpleaños() que no retorna nada hubiese otra que retornase un **int**, ¿cómo sabría entonces el compilador a cuál llamar?, Antes de continuar es preciso señalar que 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 cuenta con el método común a todos los **int**s llamado **ToString()** que lo que hace es devolver una cadena cuyos caracteres son los dígitos que forman el entero representado por el **int** sobre el que se aplica; y como la variable s es de tipo **string** (tipo que representa cadenas) es perfectamente posible almacenar dicha cadena en ella, que es lo que se hace en el código anterior.

**Creación de Objetos**

**Operador new**

Ahora que ya sabemos cómo definir las clases de objetos que podremos usar en nuestras aplicaciones ha llegado el momento de explicar cómo crear objetos de una determinada clase. La utilidad del operador **new**, que precisamente es crear objetos y cuya sintaxis es:

**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. Hay que resaltar el hecho de que **new** no devuelve el propio objeto creado, sino una referencia a la dirección de memoria dinámica donde en realidad se ha creado.

El antes comentado **constructor** de un objeto no es más que un método definido en la definición de su tipo que tiene el mismo nombre que la clase a la que pertenece el objeto y no tiene valor de retorno. Como **new** siempre devuelve una referencia a la dirección de memoria donde se cree el objeto y los constructores sólo pueden usarse como operandos de **new**, no tiene sentido que un constructor devuelva objetos, por lo que no tiene sentido incluir en su definición un campo <tipoDevuelto> y el compilador considera erróneo hacerlo (aunque se indique **void**).

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

{

string Nombre; // Campo de cada objeto Persona que almacena su nombre

int Edad; // Campo de cada objeto Persona que almacena su edad

string NIF; // Campo de cada objeto Persona que almacena su NIF

void Cumpleaños() // Incrementa en uno la edad del objeto Persona

{

Edad++;

}

Persona (string nombre, int edad, string nif) // Constructor

{

Nombre = nombre;

Edad = edad;

NIF = nif;

}

}

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 y NIF 12344321-A así:

new Persona("José", 22, "12344321-A")

Nótese que la forma en que se pasan parámetros al constructor consiste en indicar los valores que se ha de dar a cada uno de los parámetros indicados en la definición del mismo separándolos por comas. Obviamente, si un parámetro se definió como de tipo **string** habrá que pasarle una cadena, si se definió de tipo **int** habrá que pasarle un entero y, en

general, ha todo parámetro habrá que pasarle un valor de su mismo tipo (o de alguno convertible al mismo), produciéndose un error al compilar si no se hace así.

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. De ese modo, cuando creemos el objeto el compilador podrá inteligentemente determinar cuál de los constructores ha de ejecutarse en función de los valores que le pasemos al **new**.

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.

**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>()

{

}

Gracias a este constructor introducido automáticamente por el compilador, si Coche es una clase en cuya definición no se ha incluido ningún constructor, siempre será posible crear uno nuevo usando el operador **new** así:

Coche c = new Coche();

// Crea coche c llamando al constructor por defecto de Coche

**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**:

Persona (string Nombre, int Edad, string NIF)

{

this.Nombre = Nombre;

this.Edad = Edad;

this.NIF = NIF;

}

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.

**Concepto de herencia**

El mecanismo de **herencia** es uno de los pilares fundamentales en los que se basa la programación orientada a objetos.

Es un 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, nif)

{

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. Nótese además que 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

public string NIF; // Campo de cada objeto Persona que almacena su NIF

void Cumpleaños() // Incrementa en uno de edad del objeto Persona

{

Edad++;

}

public Persona (string nombre, int edad, string nif) // Constructor de Persona

{

Nombre = nombre;

Edad = edad;

NIF = nif;

}

}

class Trabajador: Persona

{

public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

Trabajador(string nombre, int edad, string nif, int sueldo):

base(nombre, edad, nif)

{ // Inicializamos cada Trabajador en base al constructor de Persona

Sueldo = sueldo;

}

public static void Main()

{

Trabajador p = new Trabajador("Josan", 22, "77588260-Z", 100000);

Console.WriteLine ("Nombre="+p.Nombre);

Console.WriteLine ("Edad="+p.Edad);

Console.WriteLine ("NIF="+p.NIF);

Console.WriteLine ("Sueldo="+p.Sueldo);

}

}

Nótese que ha sido necesario prefijar la definición de los miembros de Persona del palabra reservada **public**. Esto se debe a que por defecto los miembros de una tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo **public** conseguimos que sean accesibles desde cualquier código, como el método Main() definido en Trabajador. **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 compiladorconsiderará que éste es **:base()** Por ello hay que estar seguros de que si no se incluye **base** en la definición de algúnconstructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

Es especialmente significativo reseñar el caso de que no demos la definición de ningún constructor en la clase hija, ya que en estos casos la definición del constructor que por defecto introducirá el compilador será en realidad de la forma:

<nombreClase>()**: base(){}**

Es decir, este constructor siempre llama al constructor sin parámetros del padre del tipo que estemos definiendo, y si ése no dispone de alguno se producirá un error al compilar.

**Concepto de polimorfismo**

El **polimorfismo** es otro de los pilares fundamentales de la programación orientada a objetos. Es la capacidad de almacenar objetos de un determinado tipo en variables de tipos antecesores del primero a costa, claro está, de sólo poderse acceder a través de dicha variable a los miembros comunes a ambos tipos.

Sin embargo las versiones de los métodos virtuales a las que se llamaría a través de esas variables no serían las definidas como miembros del tipo de dichas variables, sino las definidas en el verdadero tipo de los objetos que almacenan.

A continuación se muestra un ejemplo de cómo una variable de tipo Persona puede usarse para almacenar objetos de tipo Trabajador. En esos casos el campo Sueldo del objeto referenciado por la variable no será accesible, y la versión del método Cumpleaños() a la que se podría llamar a través de la variable de tipo Persona sería la definida en la clase Trabajador, y no la definida en Persona:

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 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

{

int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

Trabajador(string nombre, int edad, string nif, int sueldo):

base(nombre, edad, nif)

{ // Inicializamos cada Trabajador en base al constructor de Persona

Sueldo = sueldo;

}

public override Cumpleaños()

{

Edad++;

Console.WriteLine("Incrementada edad de trabajador");

}

public static void Main()

{

Persona p = new Trabajador("Josan", 22, "77588260-Z", 100000);

p.Cumpleaños();

// p.Sueldo++; //ERROR: Sueldo no es miembro de Persona

}

}

El mensaje mostrado por pantalla al ejecutar este método confirma lo antes dicho respecto a que la versión de Cumpleaños() a la que se llama, ya que es:

Incrementada edad de trabajador

**Miembros de tipo**

En realidad, dentro la definición de un tipo de dato no tiene porqué incluirse sólo definiciones de miembros comunes a todos sus objetos, sino también pueden definirse miembros ligados al tipo como tal y no a los objetos del mismo. Para ello basta preceder la definición de ese miembro de la palabra reservada **static**, como muestra este ejemplo:

class A

{

int x;

static int y;

}

Los objetos de clase A sólo van a disponer del campo x, mientras que el campo y va a pertenecer a la clase A. Por esta razón se dice que los miembros con modificador **static** son **miembros de tipo** y que los no lo tienen son **miembros de** **objeto**.

Para acceder a un miembro de clase ya no es válida la sintaxis hasta ahora vista de <objeto>**.**<miembro>, pues al no estar estos miembros ligados a ningún objeto no podría ponerse nada en el campo <objeto>. La sintaxis a usar para acceder a estos miembros será <nombreClase>**.**<miembro>, como muestra ejemplo donde se asigna el valor 2 al

miembro y de la clase A definida más arriba:

A.y = 2;

Nótese que la inclusión de miembros de clase rompe con la afirmación indicada al principio del tema en la que se decía que C# es un lenguaje orientado a objetos puro en el que todo con lo que se trabaja son objetos, ya que a los miembros de tipo no se les accede a través de objetos sino nombres de tipos.

Es importante matizar que si definimos una función como **static**, entonces el código de la misma sólo podrá acceder implícitamente (sin sintaxis <objeto>.<miembro>) a otros miembros **static** del tipo de dato al que pertenezca. O sea, no se podrá acceder a ni a los miembros de objeto del tipo en que esté definido ni se podrá usar **this** ya que el método no está asociado a ningún objeto. O sea, este código sería inválido:

int x;

static void Incrementa()

{

x++; //ERROR: x es miembro de objeto e Incrementa() lo es de clase.

}

También hay que señalar que los métodos estáticos no entran dentro del mecanismo de redefiniciones descrito en este mismo tema. Dicho mecanismo sólo es aplicable a métodos de objetos, que son de quienes puede declararse variables y por tanto puede actuar el polimorifsmo. Por ello, incluir los modificadores **virtual**, **override** o **abstract** al definir un método **static** es considerado erróneo por el compilador; aunque ello no significan que los miembros **static** no se hereden, sino que sólo tiene sentido redefinirlos.

**Encapsulación**

Ya hemos visto que la herencia y el polimorfismo eran dos de los pilares fundamentales en los que es apoya la

programación orientada a objetos. Pues bien, el tercero y último es la **encapsulación**, que es un mecanismo que permite a los diseñadores de tipos de datos determinar qué miembros de los tipos creen pueden ser utilizados por otros programadores y cuáles no. Las principales ventajas que ello aporta son:

* l Se facilita a los programadores que vaya a usar el tipo de dato (programadores clientes) el aprendizaje de cómo trabajar con él, pues se le pueden ocultar todos los detalles relativos a su implementación interna y sólo dejarle visibles aquellos que puedan usar con seguridad. Además, así se les evita que cometan errores por manipular inadecuadamente miembros que no deberían tocar.
* l Se facilita al creador del tipo la posterior modificación del mismo, pues si los programadores clientes no pueden acceder a los miembros no visibles, sus aplicaciones no se verán afectadas si éstos cambian se eliminan. Gracias a esto es posible crear inicialmente tipos de datos con un diseño sencillo aunque poco eficiente, y si posteriormente es necesariomodificarlos para aumentar su eficiencia, ello puede hacerse sin afectar al código escrito en base a la no mejorada de tipo.

**public:** Puede ser accedido desde cualquier código.

**protected:** Desde una clase sólo puede accederse a miembros **protected** de objetos de esa misma clase o de subclases suyas.

**private:** Sólo puede ser accedido desde el código de la clase a la que pertenece. Es lo considerado por defecto.

**internal:** Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido.

**protected internal:** Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido o

desde clases que deriven de la clase donde se ha definido.

**Definición de variables**

Una **variable** puede verse simplemente como un almacén de objetos de un determinado tipo al que se le da un cierto nombre. Por tanto, para definir una variable sólo hay que decir cuál será el nombre que se le dará y cuál será el tipo de datos que podrá almacenar, lo que se hace con la 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 accedidas desde código ubicado dentro del método. El siguiente ejemplo muestra como definir variables de todos estos casos:

class A

{

int x, z;

int y;

void F(string a, string b)

{

Persona p;

}

}

Ya hemos visto que para crear objetos se utiliza el operador **new**. Por tanto, una forma de asignar un valor a la variable p del ejemplo anterior sería así:

Persona p;

p = new Persona("José", 22, "76543876-A");

Sin embargo, C# también proporciona una sintaxis más sencilla con la que podremos asignar un objeto a una variable en el mismo momento se define. Para ello se la ha de definir usando esta otra notación:

<tipoVariable> <nombreVariable> **=** <valorInicial>**;**

**Tipos de datos básicos**

Los **tipos de datos básicos** son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones que en C# se ha incluido una sintaxis especial para tratarlos.

**Tipo Descripción Bits Rango de valores Alias**

SByte *Bytes* con signo 8 -128 - 127 sbyte

Byte *Bytes* sin signo 8 0 - 255 byte

Int16 Enteros cortos con signo 16 [-32.768, 32.767] short

UInt16 Enteros cortos sin signo 16 [0, 65.535] ushort

Int32 Enteros normales 32 [-2.147.483.648, 2.147.483.647] int

UInt32 Enteros normales sinsigno 32 [0, 4.294.967.295] uint

Int64 Enteros largos 64 [-9.223.372.036.854.775.808, 9.223.372.036.854.775.807] long

UInt64 Enteros largos sin signo 64 [0-18.446.744.073.709.551.615] ulong

Single Reales con 7 dígitos de precisión 32 [1,5×10-45 - 3,4×1038] float

Double Reales de 15-16 dígitos de precisión 64 [5,0×10-324 - 1,7×10308] double

Decimal Reales de 28-29 dígitos de precisión 128 [1,0×10-28 - 7,9×1028] decimal

Boolean Valores lógicos 32 **true**, **false** bool

Char Caracteres Unicode 16 ['\u0000', '\uFFFF'] char

String Cadenas de caracteres Variable El permitido por la memoria string

Object Cualquier objeto Variable Cualquier objeto object

**Arreglo o Tabla unidimensional**

Una **tabla unidimensional** 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>**[]** <nombreTabla>**;**

Por ejemplo, una tabla que pueda almacenar objetos de tipo **int** se declara así:

int[] tabla;

Con esto la tabla 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 quepuede hacerse usando la siguiente sintaxis al declararla:

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

Por ejemplo, una tabla que pueda almacenar 100 objetos de tipo **int** se declara así:

int[] tabla = new int[100];

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

int[] tabla;

tabla = new int[100];

**Arreglos o Tabla multidimensional**

Una **tabla multidimensional** es una tabla cuyos elementos se encuentran organizando una estructura de varias dimensiones. 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, una tabla multidimensional 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[,] tablaMultidimensional = new int[3,4] {, , };

En realidad no es necesario indicar el número de elementos de cada dimensión de la tabla ya que pueden deducirse de los valores explícitamente indicados entre llaves, por lo que la definición anterior es similar a esta:

int[,] tablaMultidimensional = new int[,] {, , };

Incluso puede reducirse aún más la sintaxis necesaria quedando tan sólo:

int[,] tablaMultidimensional = {, , };

Si no queremos indicar explícitamente los elementos de la tabla al declararla, podemos obviarlos pero aún así indicar el tamaño de cada dimensión de la tabla (a los elementos se les daría el valor por defecto de su tipo de dato) así:

int[,] tablaMultidimensional = new int[3,4];

También podemos no especificar ni siquiera el número de elementos de la tabla de esta forma (tablaMultidimensional contendría ahora **null**):

int[,] tablaMultidimensional;

Aunque los ejemplos de tablas multidimensionales hasta ahora mostrados son de tablas de dos dimensiones, en general también es posible crear tablas de cualquier número de dimensiones. Por ejemplo, una tabla que almacene 24 elementos de tipo **int** y valor 0 en una estructura tridimensional 3x4x2 se declararía así:

int[,,] tablaMultidimensional = new int[3,4,2];

El acceso a los elementos de una tabla multidimensional es muy sencillo: sólo hay que indicar los índices de la posición que ocupe en la estructura multidimensional el elemento al que se desee acceder. Por ejemplo, para incrementar en una unidad el elemento que ocupe la posición (1,3,2) de la tabla anterior se haría (se indiza desde 0):

tablaMultidimensional[0,2,1]++;

**La clase System.Array**

En realidad, todas las tablas que definamos, sea cual sea el tipo de elementos que contengan, son objetos que derivan de **System.Array**. Es decir, van a disponer de todos los miembros que se han definido para esta clase, entre los que son destacables:

**Length**: Campo de sólo lectura que informa del número total de elementos que contiene la tabla. Si la tabla tiene más de una dimensión o nivel de anidación indica el número de elementos de todas sus dimensiones y niveles. Por ejemplo:

int[] tabla = ;

int[][] tabla2 = {new int[] , new int[] };

int[,] tabla3 = {,};

Console.WriteLine(tabla.Length); //Imprime 4

Console.WriteLine(tabla2.Length); //Imprime 5

Console.WriteLine(tabla3.Length); //Imprime 6

**Rank**: Campo de sólo lectura que almacena el número de dimensiones de la tabla.

Obviamente si la tabla no es multidimensional valdrá 1. Por ejemplo:

int[] tabla = ;

int[][] tabla2 = {new int[] , new int[] };

int[,] tabla3 = {,};

Console.WriteLine(tabla.Rank); //Imprime 1

Console.WriteLine(tabla2.Rank); //Imprime 1

Console.WriteLine(tabla3.Rank); //Imprime 2

l **int GetLength(int dimensión)**: Método que devuelve el número de elementos de la dimensión especificada. Las dimensiones se indican empezando a contar desde cero, por lo que si quiere obtenerse el número de elementos de la primera dimensión habrá que usar GetLength(0), si se quiere obtener los de la segunda habrá que usar GetLength(1), etc. Por

ejemplo:

int[,] tabla = {, };

Console.WriteLine(tabla.GetLength(0)); // Imprime 2

Console.WriteLine(tabla.GetLength(1)); // Imprime 4

l **void CopyTo(Array destino, int posición)**: Copia todos los elementos de la tabla sobre la que es aplica en la que se le pasa como primer parámetro a partir de la posición de la misma indicada como segundo parámetro. Por ejemplo:

int[] tabla1 = ;

int[] tabla2 = ;

tabla1.CopyTo(tabla2,0); // A partir de ahora, ambas tablas contienen

Ambas tablas han de ser unidimensionales. Por otro lado, y como es obvio, la tabla de destino ha de ser de un tipo que pueda almacenar los objetos de la tabla fuente, el índice especificado ha de ser válido (mayor o igual que cero y menor que el tamaño de la tabla de destino) y no ha de valer **null** ninguna.

Si no fuese así, saltarían excepciones de diversos tipos informando del error cometido (en la documentación del SDK puede ver cuáles son en concreto).

Aparte de los miembros aquí señalados, de **System.Array** cuenta con muchos más que permiten realizar tareas tan frecuentes como búsquedas de elementos, ordenaciones, etc.

**Cadenas de texto**

Una **cadena de texto** no es más que una secuencia de caracteres Unicode. En C# se representan mediante objetos del tipo tipo de dato llamado **string**, que no es más que un alias del tipo **System.String** incluido en la BCL.

Las cadenas de texto suelen crearse a partir literales de cadena o de otras cadenas previamente creadas.

Ejemplos de ambos casos se muestran a continuación:

string cadena1 = "José Antonio";

string cadena2 = cadena;

En el primer caso se ha creado un objeto **string** que representa a la cadena formada por la secuencia de caracteres José Antonio indicada literalmente (nótese que las comillas dobles entre las que se encierran los literales de cadena no forman parte del contenido de la cadena que representan sino que sólo se usan como delimitadores de la misma) En el segundo caso la variable cadena2 creada se genera a partir de la variable cadena1 ya existente, por lo que ambas variables apuntarán al mismo objeto en memoria.

Hay que tener en cuenta que el tipo **string** es un tipo referencia, por lo que en principio la comparación objetos de este tipo debería comparar sus direcciones de memoria como pasa con cualquier tipo referencia. Sin embargo, si ejecutamos el siguiente código veremos que esto no ocurre en el caso de las cadenas:

using System;

public class IgualdadCadenas

{

public static void Main()

{

string cadena1 = "José Antonio";

string cadena2 = String.Copy(cadena1);

Console.WriteLine(cadena1==cadena2);

}

}

El método **Copy()** de la clase **String** usado devuelve una copia del objeto que se le pasa como parámetro.

Por tanto, al ser objetos diferentes se almacenarán en posiciones distintas de memoria y al compararlos debería devolverse **false** como pasa con cualquier tipo referencia. Sin embargo, si ejecuta el programa verá que lo que se obtiene es precisamente lo contrario: **true**. Esto se debe a que para hacer para hacer más intuitivo el trabajo con cadenas, en C# se ha modificado el operador de igualdad para que cuando se aplique entre cadenas se considere que sus operandos son iguales sólo si son lexicográficamente equivalentes y no si referencian al mismo objeto en memoria. Además, esta comparación se hace teniendo en cuenta la capitalización usada, por lo que "Hola"=="HOLA" ó "Hola"=="hola" devolverán **false** ya que contienen las mismas letras pero con distinta capitalización.

Aparte de los métodos ya vistos, en la clase **System.String** se definen muchos otros métodos aplicables a cualquier cadena y que permiten manipularla. Los principales son:

* **int IndexOf(string subcadena)**: Indica cuál es el índice de la primera aparición de la subcadena indicada dentro de la cadena sobre la que se aplica. La búsqueda de dicha subcadena se realiza desde el principio de la cadena, pero es posible indicar en un segundo parámetro opcional de tipo **int** cuál es el índice de la misma a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la cadena sobre la que se busca, pero pasando un tercer parámetro opcional de tipo **int** es posible indicar algún índice anterior donde terminarla.

Nótese que es un método muy útil para saber si una cadena contiene o no alguna subcadena determinada, pues sólo si no la encuentra devuelve un **-1**.

* **int LastIndexOf(string subcadena)**: Funciona de forma similar a **IndexOf()** sólo que devuelve la posición de la última aparición de la subcadena buscada en lugar de devolver la de la primera.
* **string Insert(int posición, string subcadena)**: Devuelve la cadena resultante de insertar la subcadena indicada en la posición especificada de la cadena sobre la que se aplica.
* **string Remove(int posición, int número)**: Devuelve la cadena resultante de eliminar el número de caracteres indicado que hubiese en la cadena sobre al que se aplica a partir de la posición especificada.
* **string Replace(string aSustituir, string sustituta)**: Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparición de la cadena aSustituir indicada por la cadena sustituta especificada como segundo parámetro.
* **string Substring(int posición, int número)**: Devuelve la subcadena de la cadena sobre la que se aplica que comienza en la posición indicada y tiene el número de caracteres especificados. Si no se indica dicho número se devuelve la subcadena que va desde la posición indicada hasta el final de la cadena.
* **string ToUpper()** y **string ToLower()**: Devuelven, respectivamente, la cadena que resulte de convertir a mayúsculas o minúsculas la cadena sobre la que se aplican.

**Concepto de propiedad**

Una **propiedad** es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Éste código puede usarse para comprobar que no

se asignen valores inválidos, para calcular su valor sólo al solicitar su lectura, etc. Una propiedad no almacena datos, sino sólo se utiliza como si los almacenase.

Para definir una propiedad se usa la siguiente sintaxis:

< tipoPropiedad> <nombrePropiedad>

**{**

**set**

**{**

<códigoEscritura>

**}**

**get**

**{**

<códigoLectura>

**}**

**}**

Una propiedad así definida sería accedida como si de un campo de tipo <tipoPropiedad> se tratase, pero en cada lectura de su valor se ejecutaría el <códigoLectura> y en cada escritura de un valor en ella se ejecutaría <códigoEscritura>

Al escribir los bloques de código **get** y **set** hay que tener en cuenta que dentro del código **set** se puede hacer referencia al valor que se solicita asignar a través de un parámetro especial del mismo

tipo de dato que la propiedad llamado **value** (luego nosotros no podemos definir uno con ese nombre en <códigoEscritura>); y que dentro del código **get** se ha de devolver siempre un objeto del tipo de dato de la propiedad.

En realidad el orden en que aparezcan los bloques de código **set** y **get** es irrelevante. Además, es posible definir propiedades que sólo tengan el bloque **get** (**propiedades de sólo lectura**) o que sólo tengan el bloque **set** (**propiedades de sólo escritura**) Lo que no es válido es definir

propiedades que no incluyan ninguno de los dos bloques.

Las propiedades participan del mecanismo de polimorfismo igual que los métodos, siendo incluso posible definir propiedades cuyos bloques de código **get** o **set** sean abstractos. Esto se haría

prefijando el bloque apropiado con un modificador **abstract** y sustituyendo la definición de su código por un punto y coma. Por ejemplo:

using System;

abstract class A

{

public abstract int PropiedadEjemplo

{

set;

get;

}

}

class B:A

{

private int valor;

public override int PropiedadEjemplo

{

get

{

Console.WriteLine("Leído {0} de PropiedadEjemplo", valor);

return valor;

}

set

{

valor = value;

Console.WriteLine("Escrito {0} en PropiedadEjemplo", valor);

}

}

}

En este ejemplo se ve cómo se definen y redefinen propiedades abstractas. Al igual que **abstract** y **override**, también es posible usar cualquiera de los modificadores relativos a herencia ypolimorfismo ya vistos: **virtual**, **new** y **sealed**.

Nótese que aunque en el ejemplo se ha optado por asociar un campo privado valor a la propiedad PropiedadEjemplo, en realidad nada obliga a que ello se haga y es posible definir propiedades que no tenga campos asociados. Es decir, una propiedad no se tiene porqué corresponder con un

almacén de datos.

La forma de acceder a una propiedad, ya sea para lectura o escritura, es exactamente la misma que la que se usaría para acceder a un campo de su mismo tipo. Por ejemplo, se podría acceder a la propiedad de un objeto de la clase B del ejemplo anterior con:

B obj = new B();

obj.PropiedadEjemplo++;

El resultado que por pantalla se mostraría al hacer una asignación como la anterior sería:

Leído 0 de PropiedadEjemplo;

Escrito 1 en PropiedadEjemplo;

Nótese que en el primer mensaje se muestra que el valor leído es 0 porque lo que devuelve el bloque **get** de la propiedad es el valor por defecto del campo privado valor, que como es de tipo **int** tiene como valor por defecto 0.

**Instrucciones**

**Instrucción if**

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

**if (**<condición>**)**

<instruccionesIf>

*else*

*<instruccionesElse>*

El significado de esta instrucción es el siguiente: se evalúa la expresión <condición>, que ha de devolver un valor lógico. Si es cierta (devuelve **true**) se ejecutan las <instruccionesIf>, y si es falsa (**false**) 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 instrucción siguiente al **if**. En realidad, tanto <instruccionesIf> como <instruccionesElse> pueden ser una única instrucción o un bloque de instrucciones.

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

using System;

class HolaMundoIf

{

public static void Main(String[] args)

{

if (args.Length > 0)

Console.WriteLine("¡Hola {0}!", args[0]);

else

Console.WriteLine("¡Hola mundo!");

}

}

Si ejecutamos este programa sin ningún argumento veremos que el mensaje que se muestra es ¡Hola Mundo!, mientras que si lo ejecutamos con algún argumento se mostrará un mensaje de bienvenida personalizado con el primer argumento indicado.

**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** .

Los valores indicados en cada rama del **switch** han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo **char** o de tipo **string**. Además, no puede haber más de una rama con el mismo valor.

**Instrucción while**

La **instrucción while** permite ejecutar un bloque de instrucciones mientras se de 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**. En realidad <instrucciones> puede ser una única instrucción o un bloque de instrucciones.

Un ejemplo cómo utilizar esta instrucción es el siguiente:

using System;

class HolaMundoWhile

{

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!");

}

}

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

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

l **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>**);**

**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>

**Instrucciones de excepciones**

**Concepto de excepción.**

Las **excepciones** son el mecanismo recomendado en la plataforma .NET para la propagación de errores que se produzcan durante la ejecución de las aplicaciones (divisiones por cero, intentos de lectura de archivos dañados, etc.) Básicamente una excepción es un objeto derivado de **System.Exception** que se genera cuando en tiempo de ejecución se produce algún error y que contiene información sobre el mismo.

**Excepciones predefinidas comunes**

En el espacio de nombres **System** de la BCL hay predefinidas múltiples excepciones derivadas de **System.Exception** que se corresponden con los errores más comunes que pueden surgir durante la ejecución de una aplicación.

**Tipo de la excepción :**

* **ArgumentException** Pasado argumento no válido (base de excepciones de argumentos).
* **ArgumentNullException** Pasado argumento nulo.
* **ArgumentOutOfRangeException** Pasado argumento fuera de rango.
* **ArrayTypeMistmatchException** Asignación a tabla de elemento que no es de su tipo.
* **COMException** Excepción de objeto COM.
* **DivideByZeroException** División por cero.
* **IndexOutOfRangeException** Índice de acceso a elemento de tabla fuera del rango válido (menor que cero o mayor que el tamaño de la tabla).
* **InvalidCastException** Conversión explícita entre tipos no válida.
* **InvalidOperationException** Operación inválida en estado actual del objeto.
* **InteropException** Base de excepciones producidas en comunicación con código inseguro.
* **NullReferenceException** Acceso a miembro de objeto que vale **null**
* **OverflowException** Desbordamiento dentro de contexto donde se ha de comprobar los desbordamientos (expresión constante, instrucción checked, operanción

checked u opción del compilador /checked).

* **OutOfMemoryException** Falta de memoria para crear un objeto con new.
* **SEHException** Excepción SHE del API Win32.
* **StackOverflowException** Desbordamiento de la pila, generalmente debido a un excesivo número de llamadas recurrentes.
* **TypeInizializationException** Ha ocurrido alguna excepción al inicializar los campos estáticos o el constructor estático de un tipo. En InnerException se indica cuál es.

**Lanzamiento de excepciones. Instrucción throw**

Para informar de un error no basta con crear un objeto del tipo de excepción apropiado, sino que también hay pasárselo al mecanismo de propagación de excepciones del CLR. A esto se le llama **lanzar la excepción**, y para hacerlo se usa la siguiente instrucción:

**throw** <objetoExcepciónALanzar>**;**

Por ejemplo, para lanzar una excepción de tipo **DivideByZeroException** se podría hacer:

throw new DivideByZeroException();

Si el objeto a lanzar vale **null**, entonces se producirá una **NullReferenceException** que será lanzada en vez de la excepción indicada en la instrucción **throw**.

**Captura de excepciones. Instrucción try**

Una vez lanzada una excepción es posible escribir código que es encarge de tratarla. Por defecto, si este código no se escribe la excepción provoca que la aplicación aborte mostrando un mensaje de error en el que se describe la excepción producida (información de su propiedad **Message**) y dónde se ha producido (información de su propiedad **StackTrace**) Así, dado el siguiente código fuente de ejemplo:

using System;

class PruebaExcepciones

{

static void Main()

{

A obj1 = new A();

obj1.F();

}

}

class A

{

public void F()

{

G();

}

static public void G()

{

int c = 0;

int d = 2/c;

}

}

Al compilarlo no se detectará ningún error ya que al compilador no le merece la pena calcular el valor de c en tanto que es una variable, por lo que no detectará que dividir 2/c no es válido. Sin embargo, al ejecutarlo se intentará dividir por cero en esa instrucción y ello provocará que aborte la aplicación mostrando el siguiente mensaje:

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.

at PruebaExcepciones.Main()

Como se ve, en este mensaje se indica que no se ha tratado una excepción de división por cero (tipo

**DivideByZeroException**) dentro del código del método Main() del tipo PruebaExcepciones. Si al

compilar el fuente hubiésemos utilizado la opción **/debug**, el compilador habría creado un fichero **.pdb** con información extra sobre las instrucciones del ejecutable generado que permitiría que al ejecutarlo se mostrase un mensaje mucho más detallado con información sobre la instrucción exacta que provocó la excepción, la cadena de llamadas a métodos que llevaron a su ejecución y el número de línea que cada una ocupa en el fuente:

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.

at A.G() in E:\c#\Ej\ej.cs:line 22

at A.F() in E:\c#\Ej\ej.cs:line 16

at PruebaExcepciones.Main() in E:\c#\Ej\ej.cs:line 8

Si se desea tratar la excepción hay que encerrar la división dentro de una **instrucción try** con la

siguiente sintaxis:

**try**

<instrucciones>

*catch (<excepción1>)*

*<tratamiento1>*

*catch (<excepción2>)*

*<tratamiento2>*

*...*

*finally*

*<instruccionesFinally>*