1.3.8 Clase PanelDibujo

La clase que se describe en este apartado es muy importante y quizás una de las más difíciles de entender en este capítulo introductorio. La clase ***PanelDibujo*** es muy importante porque es la responsable final de que los rectángulos y círculos aparezcan dibujados en la pantalla. Esta clase deriva de la clase ***Panel***, que deriva de ***Container***, que deriva de ***Component***, que deriva de ***Object***.

Ya se ha comentado que ***Object*** es la clase más general de ***Java***. La clase ***Component*** comprende todos los objetos de ***Java*** que tienen representación gráfica, tales como botones, barras de desplazamiento, etc. Los objetos de la clase ***Container*** son objetos gráficos del ***AWT*** (*Abstract Windows Toolkit*; la librería de clases de ***Java*** que permite crear interfaces gráficas de usuario) capaces de contener otros objetos del ***AWT***. La clase ***Panel*** define los ***Container*** más sencillos, capaces de contener otros elementos gráficos (como otros paneles) y sobre la que se puede dibujar. La clase ***PanelDibujo*** contiene el código que se muestra a continuación.

```
1.    // fichero PanelDibujo.java

2.    import java.awt.*;
3.    import java.util.ArrayList;
4.    import java.util.Iterator;

5.    public class PanelDibujo extends Panel {
6.        // variable miembro
7.        private ArrayList v;

8.        // constructor
9.        public PanelDibujo(ArrayList va) {
10.           super(new FlowLayout());
11.           this.v = va;
12.       }

13.       // redefinición del método paint()
14.       public void paint(Graphics g) {
15.           Dibujable dib;
16.           Iterator it;
17.           it = v.iterator();
18.           while(it.hasNext()) {
19.               dib = (Dibujable)it.next();
20.               dib.dibujar(g);
21.           }
22.       }

23.   } // Fin de la clase PanelDibujo
```

Las sentencias 2-4 importan las clases necesarias para construir la clase ***PanelDibujo***. Se importan todas las clases del package ***java.awt***. La clase ***ArrayList*** y la interface ***Iterator*** pertenecen al package ***java.util***, y sirven para tratar colecciones o conjuntos, en este caso conjuntos de figuras dibujables.

La sentencia 5 indica que la clase ***PanelDibujo*** deriva de la clase ***Panel***, heredando de ésta y de sus super-clases ***Container*** y ***Component*** todas sus capacidades gráficas. La sentencia 7 (`private ArrayList v;`) crea una variable miembro ***v*** que es una ***referencia a un objeto*** de la clase ***ArrayList*** (nótese que no es un objeto, sino una referencia o un nombre de objeto). Las sentencias 9-12 definen el constructor de la clase, que recibe como argumento una referencia ***va*** a un objeto de la clase ***ArrayList***. En esta lista estarán almacenadas las referencias a los objetos -rectángulos y círculos- que van a ser dibujados. En la sentencia 10 (`super(new FlowLayout());`) se llama al constructor de la super-clase ***panel***, pasándole como argumento un objeto recién creado de la clase ***FlowLayout***. Como se verá más adelante al hablar de construcción de interfaces gráficas con el AWT, la clase ***FlowLayout*** se ocupa de distribuir de una determinada forma (de izquierda a derecha y de arriba

abajo) los componentes gráficos que se añaden a un "container" tal como la clase ***Panel***. En este caso no tiene mucha importancia, pero conviene utilizarlo.

Hay que introducir ahora un aspecto muy importante de ***Java*** y, en general, de la *programación orientada a objetos*. Tiene que ver con algo que es conocido con el nombre de ***Polimorfismo***. La idea básica es que ***una referencia a un objeto de una determinada clase es capaz de servir de referencia o de nombre a objetos de cualquiera de sus clases derivadas***. Por ejemplo, es posible en ***Java*** hacer lo siguiente:

```
Geometria geom1, geom2;
geom1 = new RectanguloGrafico(0, 0, 200, 100, Color.red);
geom2 = new CirculoGrafico(200, 200, 100, Color.blue);
```

Obsérvese que se han creado dos referencias de la clase ***Geometria*** que posteriormente apuntan a objetos de las clases derivadas ***RectanguloGrafico*** y ***CirculoGrafico***. Sin embargo, hay una cierta limitación en lo que se puede hacer con las referencias ***geom1*** y ***geom2***. Por ser referencias a la clase ***Geometria*** sólo se pueden utilizar las capacidades definidas en dicha clase, que se reducen a la utilización de los métodos ***perimetro()*** y ***area()***.

De la misma forma que se ha visto con la clase base ***Geometria***, en ***Java*** es posible utilizar una referencia del tipo correspondiente a una ***interface*** para manejar objetos de clases que implementan dicha ***interface***. Por ejemplo, es posible escribir:

```
Dibujable dib1, dib2;
dib1 = new RectanguloGrafico(0, 0, 200, 100, Color.red);
dib2 = new CirculoGrafico(200, 200, 100, Color.blue);
```

donde los objetos referidos por ***dib1*** y ***dib2*** pertenecen a las clases ***RectanguloGrafico*** y ***CirculoGrafico***, que implementan la interface ***Dibujable***. También los objetos ***dib1*** y ***dib2*** tienen una limitación: sólo pueden ser utilizados con los métodos definidos por la interface ***Dibujable***.

El poder utilizar nombres de una ***super-clase*** o de una ***interface*** permite tratar de un modo unificado objetos distintos, aunque pertenecientes a distintas ***sub-clases*** o bien a clases que implementan dicha ***interface***. Esta es la idea en la que se basa el ***polimorfismo***.

Ahora ya se está en condiciones de volver al código del método ***paint()***, definido en las sentencias 14-22 de la clase ***PanelDibujo***. El método ***paint()*** es un método heredado de ***Container***, que a su vez re-define el método heredado de ***Component***. En la clase ***PanelDibujo*** se da una nueva definición de este método. Una peculiaridad del método ***paint()*** es que, por lo general, el programador no tiene que preocuparse de llamarlo: se encargan de ello ***Java*** y el sistema operativo. El programador prepara por una parte la ventana y el panel en el que va a dibujar, y por otra programa en el método ***paint()*** las operaciones gráficas que quiere realizar. El sistema operativo y ***Java*** llaman a ***paint()*** cada vez que entienden que la ventana debe ser dibujada o re-dibujada. El único argumento de ***paint()*** es un objeto ***g*** de la clase ***Graphics*** que, como se ha dicho antes, constituye el ***contexto gráfico*** (color de las líneas, tipo de letra, etc.) con el que se realizarán las operaciones de dibujo.

La sentencia 15 (`Dibujable dib;`) crea una referencia de la clase ***Dibujable***, que como se ha dicho anteriormente, podrá apuntar o contener objetos de cualquier clase que implemente dicha interface. La sentencia 16 (`Iterator it;`) crea una referencia a un objeto de la interface ***Iterator*** definida en el package ***java.util***. La interface ***Iterator*** proporciona los métodos ***hasNext()***, que chequea si la colección de elementos que se está recorriendo tiene más elementos y ***next()***, que devuelve el siguiente elemento de la colección. Cualquier colección de elementos (tal como la clase ***ArrayList*** de ***Java***, o como cualquier tipo de ***lista vinculada*** programada por el usuario) puede implementar esta interface, y ser por tanto utilizada de un modo uniforme. En la sentencia 17 se

utiliza el método ***iterator()*** de la clase ***ArrayList*** (`it = v.iterator();`), que devuelve una referencia ***Iterator*** de los elementos de la lista ***v***. Obsérvese la diferencia entre el método ***iterator()*** de la clase ***ArrayList*** y la interface ***Iterator***. En ***Java*** los nombres de las clases e interfaces siempre empiezan por mayúscula, mientras que los métodos lo hacen con minúscula. Las sentencias 18-21 representan un bucle ***while*** cuyas sentencias -encerradas entre llaves {...}- se repetirán mientras haya elementos en la enumeración ***e*** (o en el vector ***v***).

La sentencia 19 (`dib = (Dibujable)it.next();`) contiene bastantes elementos nuevos e importantes. El método ***it.next()*** devuelve el siguiente objeto de la lista representada por una referencia de tipo ***Iterator***. En principio este objeto podría ser de cualquier clase. Los elementos de la clase ***ArrayList*** son referencias de la clase ***Object***, que es la clase más general de ***Java***, la clase de la que derivan todas las demás. Esto quiere decir que esas referencias pueden apuntar a objetos de cualquier clase. El nombre de la interface ***(Dibujable)*** entre paréntesis representa un ***cast*** o conversión entre tipos diferentes. En ***Java*** como en C++, la conversión entre variables u objetos de distintas clases es muy importante. Por ejemplo, ***(int)3.14*** convierte el número ***double*** 3.14 en el entero 3. Evidentemente no todas las conversiones son posibles, pero sí lo son y tienen mucho interés las conversiones entre clases que están en la misma línea jerárquica (entre ***sub-clases*** y ***super-clases***), y entre clases que implementan la misma interface. Lo que se está diciendo a la referencia ***dib*** con el ***cast*** a la interface ***Dibujable*** en la sentencia 19, es que el objeto de la enumeración va a ser tratado exclusivamente con los métodos de dicha interface. En la sentencia 20 (`dib.dibujar(g);`) se aplica el método ***dibujar()*** al objeto referenciado por ***dib***, que forma parte del iterator ***it***, obtenida a partir de la lista ***v***.

Lo que se acaba de explicar puede parecer un poco complicado, pero es típico de ***Java*** y de la programación orientada a objetos. La ventaja del método ***paint()*** así programado es que es absolutamente general: en ningún momento se hace referencia a las clases ***RectanguloGrafico*** y ***CirculoGrafico***, cuyos objetos son realmente los que se van a dibujar. Esto permite añadir nuevas clases tales como ***TrianguloGrafico***, ***PoligonoGrafico***, ***LineaGrafica***, etc., sin tener que modificar para nada el código anterior: tan sólo es necesario que dichas clases implementen la interface ***Dibujable***. Esta es una ventaja no pequeña cuando se trata de crear programas *extensibles* (que puedan crecer), *flexibles* (que se puedan modificar) y *reutilizables* (que se puedan incorporar a otras aplicaciones).

1.3.9 Clase VentanaCerrable

La clase ***VentanaCerrable*** es la última clase de este ejemplo. Es una clase de "utilidad" que mejora algo las características de la clase ***Frame*** de ***Java***, de la que deriva. La clase ***Frame*** estándar tiene una limitación y es que no responde a las acciones normales en ***Windows*** para cerrar una ventana o una aplicación (por ejemplo, clicar en la cruz de la esquina superior derecha). En ese caso, para cerrar la aplicación es necesario recurrir por ejemplo al comando ***End Task*** del ***Task Manager*** de ***Windows NT*** (que aparece con ***Ctrl+Alt+Supr***). Para evitar esta molestia se ha creado la clase ***VentanaCerrable***, que deriva de ***Frame*** e implementa la interface ***WindowListener***. A continuación se muestra el código de la clase ***VentanaCerrable***.

```
1.    // Fichero VentanaCerrable.java

2.    import java.awt.*;
3.    import java.awt.event.*;

4.    class VentanaCerrable extends Frame implements WindowListener {

5.        // constructores
6.        public VentanaCerrable() {
7.            super();
```

```
8.          }
9.          public VentanaCerrable(String title) {
10.             super(title);
11.             setSize(500,500);
12.             addWindowListener(this);
13.         }

14.         // métodos de la interface WindowsListener
15.         public void windowActivated(WindowEvent e) {;}
16.         public void windowClosed(WindowEvent e) {;}
17.         public void windowClosing(WindowEvent e) {System.exit(0);}
18.         public void windowDeactivated(WindowEvent e) {;}
19.         public void windowDeiconified(WindowEvent e) {;}
20.         public void windowIconified(WindowEvent e) {;}
21.         public void windowOpened(WindowEvent e) {;}

22.     } // fin de la clase VentanaCerrable
```

La clase ***VentanaCerrable*** contiene dos constructores. El primero de ellos es un constructor por defecto (sin argumentos) que se limita a llamar al constructor de la super-clase ***Frame*** con la palabra ***super***. El segundo constructor admite un argumento para poner título a la ventana; llama también al constructor de ***Frame*** pasándole este mismo argumento. Después establece un tamaño para la ventana creada (el tamaño por defecto para ***Frame*** es cero).

La sentencia 12 (`addWindowListener(this);`**)** es muy importante y significativa sobre la forma en que el AWT de ***Java*** gestiona los ***eventos*** sobre las ventanas y en general sobre lo que es la interface gráfica de usuario. Cuando un elemento gráfico -en este caso la ventana- puede recibir eventos del usuario es necesario indicar quién se va a encargar de procesar esos eventos. De ordinario al producirse un evento se debe activar un método determinado que se encarga de procesarlo y realizar las acciones pertinentes (en este caso cerrar la ventana y la aplicación). La sentencia 12 ejecuta el método ***addWindowListener()*** de la clase ***Frame*** (que a su vez lo ha heredado de la clase ***Window***). El argumento que se le pasa a este método indica qué objeto se va a responsabilizar de gestionar los eventos que reciba la ventana implementando la interface ***WindowListener***. En este caso, como el argumento que se le pasa es ***this***, la propia clase ***VentanaCerrable*** debe ocuparse de gestionar los eventos que reciba. Así es, puesto que dicha clase implementa la interface ***WindowListener*** según se ve en la sentencia 4. Puede notarse que como el constructor por defecto de las sentencias 6-8 no utiliza el método ***addWindowListener()***, si se construye una ***VentanaCerrable*** sin título no podrá ser cerrada del modo habitual. Así se ha hecho deliberadamente en este ejemplo para que el lector lo pueda comprobar con facilidad.

La interface ***WindowListener*** define los siete métodos necesarios para gestionar los siete eventos con los que se puede actuar sobre una ventana. Para cerrar la ventana sólo es necesario definir el método ***windowClosing()***. Sin embargo, el implementar una interface obliga siempre a definir todos sus métodos. Por ello en las sentencias 15-21 todos los métodos están vacíos (solamente el punto y coma entre llaves), excepto el que realmente interesa, que llama al método ***exit()*** de la clase ***System***. El argumento “0” indica terminación normal del programa.

1.3.10 Consideraciones adicionales sobre el Ejemplo1

Es muy importante entender los conceptos explicados; esto puede facilitar mucho la comprensión de los capítulos que siguen.

Se puede practicar con este ejemplo creando algunos objetos más en el programa principal o introduciendo alguna otra pequeña modificación.

1.4 NOMENCLATURA HABITUAL EN LA PROGRAMACIÓN EN JAVA

Los nombres de ***Java*** son sensibles a las letras mayúsculas y minúsculas. Así, las variables ***masa***, ***Masa*** y ***MASA*** son consideradas variables completamente diferentes. Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de los programas de ordenador. Se recomienda seguir las siguientes instrucciones:

1. En ***Java*** es habitual utilizar nombres con minúsculas, con las excepciones que se indican en los puntos siguientes.
2. Cuando un nombre consta de varias palabras es habitual poner una a continuación de otra, poniendo con mayúscula la primera letra de la palabra que sigue a otra (Ejemplos: *elMayor()*, *VentanaCerrable*, *RectanguloGrafico*, *addWindowListener()*).
3. Los nombres de ***clases*** e ***interfaces*** comienzan siempre por mayúscula (Ejemplos: *Geometria*, *Rectangulo*, *Dibujable*, *Graphics*, *ArrayList*, *Iterator*).
4. Los nombres de ***objetos***, los nombres de ***métodos*** y ***variables miembro***, y los nombres de las ***variables locales*** de los métodos, comienzan siempre por minúscula (Ejemplos: *main()*, *dibujar()*, *numRectangulos*, *x*, *y*, *r*).
5. Los nombres de las ***variables finales***, es decir de las constantes, se definen siempre con mayúsculas (Ejemplo: *PI*)

1.5 ESTRUCTURA GENERAL DE UN PROGRAMA JAVA

El anterior ejemplo presenta la estructura habitual de un programa realizado en cualquier lenguaje ***orientado a objetos*** u ***OOP*** (*Object Oriented Programming*), y en particular en el lenguaje ***Java***. Aparece una clase que contiene el programa principal (aquel que contiene la función ***main()***) y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa principal. Los ficheros fuente tienen la extensión ****.java***, mientras que los ficheros compilados tienen la extensión ****.class***.

Un fichero fuente (****.java***) puede contener más de una clase, pero sólo una puede ser ***public***. El nombre del fichero fuente debe coincidir con el de la clase ***public*** (con la extensión ****.java***). Si por ejemplo en un fichero aparece la declaración (***public class MiClase {...}***) entonces el nombre del fichero deberá ser ***MiClase.java***. Es importante que coincidan mayúsculas y minúsculas ya que ***MiClase.java*** y ***miclase.java*** serían clases diferentes para ***Java***. Si la clase no es ***public***, no es necesario que su nombre coincida con el del fichero. Una clase puede ser ***public*** o ***package*** (default), pero no ***private*** o ***protected***. Estos conceptos se explican posteriormente.

De ordinario una aplicación está constituida por varios ficheros ****.class***. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función ***main()*** (sin la extensión ****.class***). Las clases de ***Java*** se agrupan en ***packages***, que son librerías de clases. Si las clases no se definen como pertenecientes a un ***package***, se utiliza un ***package*** por defecto (*default*) que es el directorio activo. Los ***packages*** se estudian con más detenimiento el Apartado 3.6, a partir de la página 44.

1.5.1 Concepto de Clase

Una ***clase*** es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos. A estos datos y funciones pertenecientes a una clase se les denomina ***variables*** y ***métodos*** o ***funciones miembro***. La programación orientada a objetos se basa en la programación de clases. Un programa se construye a partir de un conjunto de clases.

Una vez definida e implementada una clase, es posible declarar elementos de esta clase de modo similar a como se declaran las variables del lenguaje (de los tipos primitivos *int, double, String, …*). Los elementos declarados de una clase se denominan ***objetos*** de la clase. De una única clase se pueden declarar o crear numerosos ***objetos***. La ***clase*** es lo genérico: es el patrón o modelo para crear ***objetos***. Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores, en general distintos de los demás objetos de la clase. Las clases pueden tener variables ***static***, que son propias de la clase y no de cada objeto.

1.5.2 Herencia

La herencia permite que se pueden definir nuevas clases basadas en clases existentes, lo cual facilita re-utilizar código previamente desarrollado. Si una clase deriva de otra (***extends***) hereda todas sus variables y métodos. La clase derivada puede ***añadir*** nuevas variables y métodos y/o ***redefinir*** las variables y métodos heredados.

En ***Java***, a diferencia de otros lenguajes orientados a objetos, una clase sólo puede derivar de una única clase, con lo cual no es posible realizar ***herencia múltiple*** en base a clases. Sin embargo es posible "simular" la herencia múltiple en base a las ***interfaces***.

1.5.3 Concepto de Interface

Una ***interface*** es un conjunto de declaraciones de funciones. Si una clase implementa (***implements***) una ***interface***, debe definir ***todas*** las funciones especificadas por la ***interface***. Una ***clase*** puede implementar más de una ***interface***, representando una forma alternativa de la herencia múltiple.

A su vez, una ***interface*** puede derivar de otra o incluso de varias ***interfaces***, en cuyo caso incorpora todos los métodos de las ***interfaces*** de las que deriva.

1.5.4 Concepto de Package

Un ***package*** es una agrupación de clases. Existen una serie de ***packages*** incluidos en el lenguaje (ver jerarquía de clases que aparece en el ***API*** de ***Java***).

Además el usuario puede crear sus propios ***packages***. Lo habitual es juntar en ***packages*** las clases que estén relacionadas. Todas las clases que formen parte de un ***package*** deben estar en el mismo directorio.

1.5.5 La jerarquía de clases de Java (API)

Durante la generación de código en ***Java***, es recomendable y casi necesario tener siempre a la vista la documentación ***on-line*** del ***API*** de ***Java 1.1*** ó ***Java 1.2***. En dicha documentación es posible ver tanto la jerarquía de clases, es decir la relación de herencia entre clases, como la información de los distintos ***packages*** que componen las librerías base de ***Java***.

Es importante distinguir entre lo que significa ***herencia*** y ***package***. Un ***package*** es una agrupación arbitraria de clases, una forma de organizar las clases. La ***herencia*** sin embargo consiste

en crear nuevas clases en base a otras ya existentes. Las clases incluidas en un *package* ***no derivan*** por lo general de una única clase.

En la documentación ***on-line*** se presentan ambas visiones: "***Package Index***" y "***Class Hierarchy***", tanto en ***Java 1.1*** como en ***Java 1.2***, con pequeñas variantes. La primera presenta la estructura del ***API*** de ***Java*** agrupada por ***packages***, mientras que en la segunda aparece la jerarquía de clases. Hay que resaltar una vez más el hecho de que todas las clases en ***Java*** son derivadas de la clase ***java.lang.Object***, por lo que heredan todos los métodos y variables de ésta.

Si se selecciona una clase en particular, la documentación muestra una descripción detallada de todos los métodos y variables de la clase. A su vez muestra su herencia completa (partiendo de la clase ***java.lang.Object***).

2. PROGRAMACIÓN EN JAVA

En este capítulo se presentan las características generales de ***Java*** como lenguaje de programación algorítmico. En este apartado ***Java*** es muy similar a ***C/C++***, lenguajes en los que está inspirado. Se va a intentar ser breve, considerando que el lector ya conoce algunos otros lenguajes de programación y está familiarizado con lo que son variables, bifurcaciones, bucles, etc.

2.1 VARIABLES

Una ***variable*** es un ***nombre*** que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en ***Java*** hay dos tipos principales de variables:

1. Variables de ***tipos primitivos***. Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. ***Java*** permite distinta precición y distintos rangos de valores para estos tipos de variables (***char***, ***byte***, ***short***, ***int***, ***long***, ***float***, ***double***, ***boolean***). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.
2. Variables ***referencia***. Las variables referencia son referencias o nombres de una información más compleja: ***arrays*** u ***objetos*** de una determinada clase.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

1. Variables ***miembro*** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser *tipos primitivos* o *referencias*.
2. Variables ***locales***: Se definen dentro de un método o más en general *dentro de cualquier bloque* entre ***llaves*** {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también *tipos primitivos* o *referencias*.

2.1.1 Nombres de Variables

Los nombres de variables en ***Java*** se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por ***Java*** como operadores o separadores (,.+-*/ etc.).

Existe una serie de ***palabras reservadas*** las cuales tienen un significado especial para ***Java*** y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

abstract	boolean	break	byte	case	catch
char	class	const*	continue	default	do
double	else	extends	final	finally	float
for	goto*	if	implements	import	instanceof
int	interface	long	native	new	null
package	private	protected	public	return	short
static	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje ***Java***.

2.1.2 Tipos Primitivos de Variables

Se llaman ***tipos primitivos*** de variables de ***Java*** a aquellas variables sencillas que contienen los tipos de información más habituales: valores *boolean*, *caracteres* y *valores numéricos* enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores ***true*** y ***false*** (***boolean***); un tipo para almacenar caracteres (***char***), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (***byte***, ***short***, ***int*** y ***long***) y dos para valores reales de punto flotante (***float*** y ***double***). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la Tabla 2.1.

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 2.1. Tipos primitivos de variables en Java.

Los tipos primitivos de ***Java*** tienen algunas características importantes que se resumen a continuación:

1. El tipo ***boolean*** no es un valor numérico: sólo admite los valores ***true*** o ***false***. El tipo ***boolean*** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser ***boolean***.
2. El tipo ***char*** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.
3. Los tipos ***byte***, ***short***, ***int*** y ***long*** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en ***Java*** no hay enteros ***unsigned***.
4. Los tipos ***float*** y ***double*** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.
5. Se utiliza la palabra ***void*** para indicar la ausencia de un tipo de variable determinado.
6. A diferencia de C/C++, los tipos de variables en ***Java*** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un ***int*** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.
7. Existen extensiones de ***Java 1.2*** para aprovechar la arquitectura de los procesadores ***Intel***, que permiten realizar operaciones de punto flotente con una precisión extendida de 80 bits.

2.1.3 Cómo se definen e inicializan las variables

Una variable se define especificando el ***tipo*** y el ***nombre*** de dicha variable. Estas variables pueden ser tanto de tipos ***primitivos*** como ***referencias*** a objetos de alguna clase perteneciente al ***API*** de ***Java*** o generada por el usuario. Si no se especifica un valor en su declaración, las variable

primitivas se inicializan a cero (salvo *boolean* y *char*, que se inicializan a ***false*** y '\0'). Análogamente las variables de tipo ***referencia*** son inicializadas por defecto a un valor especial: ***null***.

Es importante distinguir entre la ***referencia*** a un objeto y el ***objeto*** mismo. Una ***referencia*** es una variable que indica dónde está guardado un objeto en la memoria del ordenador (a diferencia de C/C++, ***Java*** no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los ***punteros***). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor ***null***. Si se desea que esta ***referencia*** apunte a un nuevo objeto es necesario crear el objeto utilizando el operador ***new***. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la ***referencia*** declarada a otra referencia a un objeto existente previamente.

Un tipo particular de referencias son los ***arrays*** o vectores, sean éstos de variables primitivas (por ejemplo, un vector de enteros) o de objetos. En la declaración de una referencia de tipo ***array*** hay que incluir los ***corchetes*** []. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos ***MyClass***. ***Java*** garantiza que los elementos del vector son inicializados a ***null*** o a cero (según el tipo de dato) en caso de no indicar otro valor.

```
Ejemplos de declaración e inicialización de variables:
int x;               // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;           // Declaración de la variable primitiva y. Se inicializa a 5

MyClass unaRef;                // Declaración de una referencia a un objeto MyClass.
                               // Se inicializa a null
unaRef = new MyClass();        // La referencia "apunta" al nuevo objeto creado
                               // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef;   // Declaración de una referencia a un objeto MyClass.
                               // Se inicializa al mismo valor que unaRef
int [] vector;        // Declaración de un array. Se inicializa a null
vector = new int[10]; // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1};// Declaración e inicialización de un vector de 3
                               // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5];// Se crea un vector de 5 referencias a objetos
                               // Las 5 referencias son inicializadas a null
lista[0] = unaRef;             // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass();      // Se asigna a lista[1] la referencia al nuevo objeto
                               // El resto (lista[2]…lista[4] siguen con valor null
```

En el ejemplo mostrado las referencias ***unaRef***, ***segundaRef*** y ***lista[0]*** actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

2.1.4 Visibilidad y vida de las variables

Se entiende por ***visibilidad***, ***ámbito*** o ***scope*** de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en una expresión. En ***Java*** todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves **{}**, es decir dentro de un ***bloque***, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque ***if*** no serán válidas al finalizar las sentencias correspondientes a dicho ***if*** y las variables miembro de una ***clase*** (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Las variables miembro de una clase declaradas como ***public*** son accesibles a través de una ***referencia*** a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como ***private*** no son accesibles directamente desde otras clases. Las ***funciones miembro***

de una clase tienen acceso directo a ***todas*** las variables miembro de la clase sin necesidad de anteponer el nombre de un objeto de la clase. Sin embargo las funciones miembro de una clase ***B*** derivada de otra ***A***, tienen acceso a todas las variables miembro de ***A*** declaradas como ***public*** o ***protected***, pero no a las declaradas como ***private***. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como ***public*** o ***protected***. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local que ya existiera. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador ***this***, en la forma ***this.varname***.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. En ***Java*** la forma de crear nuevos ***objetos*** es utilizando el operador ***new***. Cuando se utiliza el operador ***new***, la variable de tipo ***referencia*** guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo ***referencia*** es apuntado. La eliminación de los objetos la realiza el programa denominado ***garbage collector***, quien automáticamente libera o borra la memoria ocupada por un ***objeto*** cuando no existe ninguna ***referencia*** apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

2.1.5 Casos especiales: Clases BigInteger y BigDecimal

Java 1.1 incorporó dos nuevas clases destinadas a operaciones aritméticas que requieran gran precisión: ***BigInteger*** y ***BigDecimal***. La forma de operar con objetos de estas clases difiere de las operaciones con variables primitivas. En este caso hay que realizar las operaciones utilizando métodos propios de estas clases (***add()*** para la suma, ***subtract()*** para la resta, ***divide()*** para la división, etc.). Se puede consultar la ayuda sobre el package ***java.math***, donde aparecen ambas clases con todos sus métodos.

Los objetos de tipo ***BigInteger*** son capaces de almacenar cualquier número entero sin perder información durante las operaciones. Análogamente los objetos de tipo ***BigDecimal*** permiten trabajar con el número de decimales deseado.

2.2 OPERADORES DE JAVA

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes.

2.2.1 Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: ***suma*** (+), ***resta*** (-), ***multiplicación*** (*), ***división*** (/) y ***resto de la división*** (%).

2.2.2 Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el ***operador igual*** (=). La forma general de las sentencias de asignación con este operador es:

```
variable = expression;
```

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Tabla 2.2. Otros operadores de asignación.

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La Tabla 2.2 muestra estos operadores y su equivalencia con el uso del ***operador igual*** (=).

2.2.3 Operadores unarios

Los operadores ***más*** (+) y ***menos*** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en ***Java*** es el estándar de estos operadores.

2.2.4 Operador instanceof

El operador ***instanceof*** permite saber si un objeto pertenece o no a una determinada clase. Es un operador binario cuya forma general es,

```
objectName instanceof ClassName
```

y que devuelve ***true*** o ***false*** según el objeto pertenezca o no a la clase.

2.2.5 Operador condicional ?:

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

```
booleanExpression ? res1 : res2
```

donde se evalúa ***booleanExpression*** y se devuelve ***res1*** si el resultado es ***true*** y ***res2*** si el resultado es ***false***. Es el único operador ternario (tres argumentos) de ***Java***. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

```
x=1 ;  y=10;  z = (x<y)?x+3:y+8;
```

asignarían a ***z*** el valor 4, es decir x+3.

2.2.6 Operadores incrementales

Java dispone del operador ***incremento*** (++) y ***decremento*** (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. *Precediendo a la variable* (por ejemplo: ***++i***). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
2. *Siguiendo a la variable* (por ejemplo: ***i++***). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más

complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles ***for*** es una de las aplicaciones más frecuentes de estos operadores.

2.2.7 Operadores relacionales

Los ***operadores relacionales*** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor ***boolean*** (***true*** o ***false***) según se cumpla o no la relación considerada. La Tabla 2.3 muestra los operadores relacionales de ***Java***.

Operador	Utilización	El resultado es true
`>`	`op1 > op2`	si op1 es mayor que op2
`>=`	`op1 >= op2`	si op1 es mayor o igual que op2
`<`	`op1 < op2`	si op1 es menor que op2
`<=`	`op1 <= op2`	si op1 es menor o igual que op2
`==`	`op1 == op2`	si op1 y op2 son iguales
`!=`	`op1 != op2`	si op1 y op2 son diferentes

Tabla 2.3. Operadores relacionales.

Estos operadores se utilizan con mucha frecuencia en las ***bifurcaciones*** y en los ***bucles***, que se verán en próximos apartados de este capítulo.

2.2.8 Operadores lógicos

Los operadores lógicos se utilizan para construir ***expresiones lógicas***, combinando valores lógicos (***true*** y/o ***false***) o los resultados de los operadores ***relacionales***. La Tabla 2.4 muestra los operadores lógicos de ***Java***. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser ***true*** y el primero es ***false***, ya se sabe que la condición de que ambos sean ***true*** no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
`&&`	`AND`	`op1 && op2`	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
`||`	`OR`	`op1 || op2`	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
`!`	`negación`	`! op`	true si op es false y false si op es true
`&`	`AND`	`op1 & op2`	true si op1 y op2 son true. Siempre se evalúa op2
`|`	`OR`	`op1 | op2`	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 2.4. Operadores lógicos.

2.2.9 Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método ***println()***. La variable numérica ***result*** es convertida automáticamente por ***Java*** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

2.2.10 Operadores que actúan a nivel de bits

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o ***flags***, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla 2.5 muestra los operadores de ***Java*** que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	~op2	Operador complemento (invierte el valor de cada bit)

Tabla 2.5. Operadores a nivel de bits.

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable ***flags*** con los bits activados que se deseen. Por ejemplo, para construir una variable ***flags*** que sea 00010010 bastaría hacer ***flags***=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

```
if (flags & 2 == 2) {...}
```

La Tabla 2.6 muestra los operadores de asignación a nivel de bits.

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Tabla 2.6. Operadores de asignación a nivel de bits.

2.2.11 Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de *x/y*z* depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de ***mayor a menor*** precedencia:

postfix operators	`[] . (params) expr++ expr--`
unary operators	`++expr --expr +expr -expr ~ !`
creation or cast	`new (type)expr`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`

relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
conditional	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

En ***Java***, todos los operadores binarios, excepto los operadores de asignación, se evalúan de ***izquierda a derecha***. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

2.3 ESTRUCTURAS DE PROGRAMACIÓN

En este apartado se supone que el lector tiene algunos conocimientos de programación y por lo tanto no se explican en profundidad los conceptos que aparecen.

Las ***estructuras de programación*** o ***estructuras de control*** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados ***bifurcaciones*** y ***bucles***. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a *concepto*, aunque su *sintaxis* varía de un lenguaje a otro. La sintaxis de ***Java*** coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

2.3.1 Sentencias o expresiones

Una ***expresión*** es un conjunto variables unidos por ***operadores***. Son órdenes que se le dan al computador para que realice una tarea determinada.

Una ***sentencia*** es una ***expresión*** que acaba en ***punto y coma*** (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

2.3.2 Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de ***Java*** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras "//" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos **/***…***/**. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de cada línea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado de líneas
   ya que sólo requiere modificar
   el comienzo y el final. */
```

En ***Java*** existe además una forma especial de introducir los comentarios (utilizando /**…*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las ***clases*** y ***packages*** desarrollados por el programador. Una vez introducidos los comentarios, el programa ***javadoc.exe*** (incluido en el ***JDK***) genera de forma automática la información de forma similar a la presentada en la propia documentación del ***JDK***. La sintaxis de estos comentarios y la forma de utilizar el programa ***javadoc.exe*** se puede encontrar en la información que viene con el ***JDK***.

2.3.3 Bifurcaciones

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el *flujo de ejecución* de un programa. Existen dos bifurcaciones diferentes: ***if*** y ***switch***.

2.3.3.1 Bifurcación if

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor ***true***). Tiene la forma siguiente:

```
if (booleanExpression) {
    statements;
}
```

Las ***llaves*** {} sirven para agrupar en un ***bloque*** las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del ***if***.

2.3.3.2 Bifurcación if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el ***else*** se ejecutan en el caso de no cumplirse la expresión de comparación (***false***),

```
if (booleanExpression) {
    statements1;
} else {
    statements2;
}
```

2.3.3.3 Bifurcación if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al ***else***.

```
if (booleanExpression1) {
    statements1;
} else if (booleanExpression2) {
    statements2;
} else if (booleanExpression3) {
    statements3;
} else {
    statements4;
}
```

Véase a continuación el siguiente ejemplo:

```
int numero = 61;                  // La variable "numero" tiene dos dígitos
if(Math.abs(numero) < 10)         // Math.abs() calcula el valor absoluto.  (false)
    System.out.println("Numero tiene 1 digito ");
else if (Math.abs(numero) < 100) // Si numero es 61, estamos en este caso (true)
    System.out.println("Numero tiene 1 digito ");
else {                            // Resto de los casos
    System.out.println("Numero tiene mas de 3 digitos ");
    System.out.println("Se ha ejecutado la opcion por defecto ");
}
```

2.3.3.4 Sentencia switch

Se trata de una alternativa a la bifurcación ***if elseif else*** cuando se compara la ***misma expresión*** con distintos valores. Su forma general es la siguiente:

```
switch (expression) {
    case value1:  statements1; break;
    case value2:  statements2; break;
    case value3:  statements3; break;
    case value4:  statements4; break;
    case value5:  statements5; break;
    case value6:  statements6; break;
    [default: statements7;]
}
```

Las características más relevantes de ***switch*** son las siguientes:

1. Cada sentencia ***case*** se corresponde con un único valor de ***expression***. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos. El ejemplo del Apartado 2.3.3.3 no se podría realizar utilizando ***switch***.
2. Los valores no comprendidos en ninguna sentencia ***case*** se pueden gestionar en ***default***, que es opcional.
3. En ausencia de ***break***, cuando se ejecuta una sentencia ***case*** se ejecutan también todas las ***case*** que van a continuación, hasta que se llega a un ***break*** o hasta que se termina el ***switch***.

   Ejemplo:

```
char c = (char)(Math.random()*26+'a'); // Generación aleatoria de letras minúsculas
System.out.println("La letra " + c );
switch (c) {
   case 'a': // Se compara con la letra a
   case 'e': // Se compara con la letra e
   case 'i': // Se compara con la letra i
   case 'o': // Se compara con la letra o
   case 'u': // Se compara con la letra u
      System.out.println(" Es una vocal "); break;
   default:
      System.out.println(" Es una consonante ");
}
```

2.3.4 Bucles

Un ***bucle*** se utiliza para realizar un proceso repetidas veces. Se denomina también ***lazo*** o ***loop***. El código incluido entre las ***llaves*** {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (***booleanExpression***) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

2.3.4.1 Bucle while

Las sentencias ***statements*** se ejecutan mientras ***booleanExpression*** sea ***true***.

```
while (booleanExpression) {
    statements;
}
```

2.3.4.2 Bucle for

La forma general del bucle ***for*** es la siguiente:

```
for (initialization; booleanExpression; increment) {
    statements;
}
```

que es equivalente a utilizar ***while*** en la siguiente forma,

```
initialization;
while (booleanExpression) {
    statements;
    increment;
}
```

La sentencia o sentencias ***initialization*** se ejecuta al comienzo del ***for***, e ***increment*** después de ***statements***. La ***booleanExpression*** se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor ***false***. Cualquiera de las tres partes puede estar vacía. La ***initialization*** y el ***increment*** pueden tener varias expresiones separadas por comas.

Por ejemplo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:

```
for(int i = 1, j = i + 10;  i < 5;  i++, j = 2*i) {
    System.out.println(" i = " + i + " j = " + j);
}
```

Salida:

```
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

2.3.4.3 Bucle do while

Es similar al bucle ***while*** pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los ***statements***, se evalúa la condición: si resulta ***true*** se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a ***false*** finaliza el bucle. Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

```
do {
    statements
} while (booleanExpression);
```

2.3.4.4 Sentencias break y continue

La sentencia ***break*** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando, sin sin realizar la ejecución del resto de las sentencias.

La sentencia ***continue*** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración “i” que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

2.3.4.5 Sentencias break y continue con etiquetas

Las ***etiquetas*** permiten indicar un lugar donde continuar la ejecución de un programa después de un ***break*** o ***continue***. El único lugar donde se pueden incluir etiquetas es ***justo delante de un bloque*** de código entre llaves {} (*if, switch, do...while, while, for*) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (*break*) o continuar con la siguiente iteración (*continue*) de un bucle que no es el actual.

Por tanto, la sentencia ***break labelName*** finaliza el bloque que se encuentre a continuación de ***labelName***. Por ejemplo, en las sentencias,

```
bucleI:   // etiqueta o label
for ( int i = 0, j = 0; i < 100; i++){
    while ( true ) {
        if( (++j) > 5) { break bucleI; }   // Finaliza ambos bucles
        else { break; }                    // Finaliza el bucle interior (while)
    }
}
```

la expresión `break bucleI;` finaliza los dos bucles simultáneamente, mientras que la expresión `break;` sale del bucle ***while*** interior y seguiría con el bucle ***for*** en ***i***. Con los valores presentados ambos bucles finalizarán con i = 5 y j = 6 (se invita al lector a comprobarlo).

La sentencia ***continue*** (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

```
continue bucle1;
```

transfiere el control al bucle ***for*** que comienza después de la etiqueta ***bucle1:*** para que realice una nueva iteración, como por ejemplo:

```
bucle1:
for (int i=0; i<n; i++) {
    bucle2:
    for (int j=0; j<m; j++) {
        ...
        if (expression) continue bucle1; then continue bucle2;
        ...
    }
}
```

2.3.4.6 Sentencia return

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia ***return***. A diferencia de ***continue*** o ***break***, la sentencia ***return*** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (`return value;`).

2.3.4.7 Bloque try {...} catch {...} finally {...}

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje ***Java***, una ***Exception*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas ***excepciones*** son ***fatales*** y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras ***excepciones***, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser ***recuperables***. En este caso el

programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo un nuevo ***path*** del fichero no encontrado).

Los errores se representan mediante clases derivadas de la clase ***Throwable***, pero los que tiene que chequear un programador derivan de ***Exception*** (***java.lang.Exception*** que a su vez deriva de ***Throwable***). Existen algunos tipos de excepciones que ***Java*** obliga a tener en cuenta. Esto se hace mediante el uso de bloques ***try***, ***catch*** y ***finally***.

El código dentro del bloque ***try*** está "vigilado". Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque ***catch***, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques ***catch*** como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque ***finally***, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

En el caso en que el código de un método pueda generar una ***Exception*** y no se desee incluir en dicho método la gestión del error (es decir los bucles ***try/catch*** correspondientes), es necesario que el método pase la ***Exception*** al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra ***throws*** seguida del nombre de la ***Exception*** concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques ***try/catch*** o volver a pasar la ***Exception***. De esta forma se puede ir pasando la ***Exception*** de un método a otro hasta llegar al último método del programa, el método ***main()***.

En el siguiente ejemplo se presentan dos métodos que deben "controlar" una ***IOException*** relacionada con la lectura ficheros y una ***MyException*** propia. El primero de ellos (*metodo1*) realiza la gestión de las excepciones y el segundo (*metodo2*) las pasa al siguiente método.

```
void metodo1() {
    ...
    try {
        ... // Código que puede lanzar las excepciones IOException y MyException
    } catch (IOException e1) {// Se ocupa de IOException simplemente dando aviso
        System.out.println(e1.getMessage());
    } catch (MyException e2) {
        // Se ocupa de MyException dando un aviso y finalizando la función
        System.out.println(e2.getMessage()); return;
    } finally { // Sentencias que se ejecutarán en cualquier caso
        ...
    }]
    ...
} // Fin del metodo1

void metodo2() throws IOException, MyException {
    ...
    // Código que puede lanzar las excepciones IOException y MyException
    ...
} // Fin del metodo2
```

El tratamiento de ***excepciones*** se desarrollará con más profundidad en el Capítulo 8, a partir de la página 135.