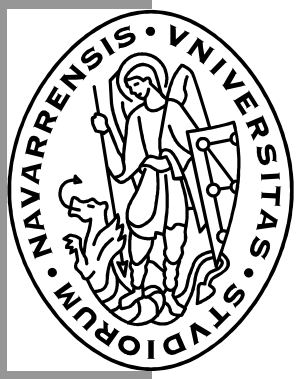


tecnun

CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda Informática ...

Aprenda Java

como si estuviera en primero

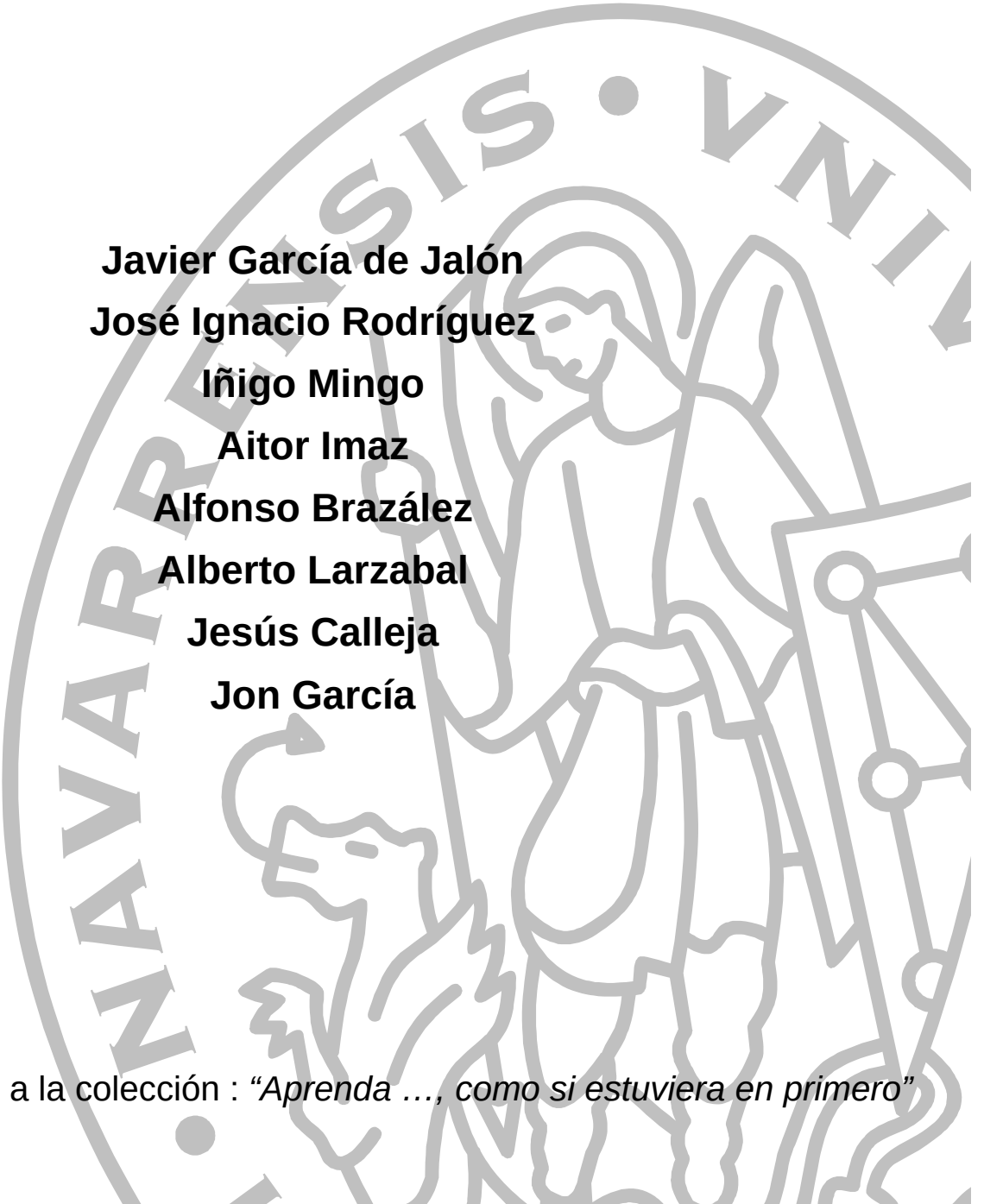
San Sebastián, Enero 2000

Javier García de Jalón • José Ignacio Rodríguez • Iñigo Mingo • Aitor Imaz
Alfonso Brazález • Alberto Larzabal • Jesús Calleja • Jon García

tecnun
CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda Java

como si estuviera en primero



Javier García de Jalón
José Ignacio Rodríguez
Iñigo Mingo
Aitor Imaz
Alfonso Brazález
Alberto Larzabal
Jesús Calleja
Jon García

Perteneciente a la colección : *“Aprenda …, como si estuviera en primero”*

ÍNDICE

1. INTRODUCCIÓN A JAVA

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una "*máquina hipotética o virtual*" denominada ***Java Virtual Machine*** (***JVM***). Era la ***JVM*** quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "*Write Once, Run Everywhere*". A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadores, ***Java*** se introdujo a finales de 1995. La clave fue la incorporación de un intérprete ***Java*** en la versión 2.0 del programa Netscape Navigator, produciendo una verdadera revolución en Internet. ***Java 1.1*** apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. ***Java 1.2***, más tarde rebautizado como ***Java 2***, nació a finales de 1998.

Al programar en ***Java*** no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de ***clases*** preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el ***API*** o ***Application Programming Interface*** de ***Java***). ***Java*** incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.). Por eso muchos expertos opinan que ***Java*** es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo del lenguaje ***Java*** es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor de ***Web***, en una base de datos o en cualquier otro lugar.

Java es un lenguaje muy completo (de hecho se está convirtiendo en un macro-lenguaje: ***Java 1.0*** tenía 12 packages; ***Java 1.1*** tenía 23 y ***Java 1.2*** tiene 59). En cierta forma casi todo depende de casi todo. Por ello, conviene aprenderlo de modo *iterativo*: primero una visión muy general, que se va refinando en sucesivas iteraciones. Una forma de hacerlo es empezar con un ejemplo completo en el que ya aparecen algunas de las características más importantes.

La compañía ***Sun*** describe el lenguaje ***Java*** como "*simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico*". Además de una serie de halagos por parte de ***Sun*** hacia su propia criatura, el hecho es que todo ello describe bastante bien el lenguaje ***Java***, aunque en algunas de esas características el

lenguaje sea todavía bastante mejorable. Algunas de las anteriores ideas se irán explicando a lo largo de este manual.

1.1 QUÉ ES JAVA 2

Java 2 (antes llamado ***Java 1.2*** o ***JDK 1.2***) es la tercera versión importante del lenguaje de programación ***Java***.

No hay cambios conceptuales importantes respecto a ***Java 1.1*** (en ***Java 1.1*** sí los hubo respecto a ***Java 1.0***), sino extensiones y ampliaciones, lo cual hace que a muchos efectos –por ejemplo, para esta introducción- sea casi lo mismo trabajar con ***Java 1.1*** o con ***Java 1.2***.

Los programas desarrollados en ***Java*** presentan diversas ventajas frente a los desarrollados en otros lenguajes como C/C++. La ejecución de programas en ***Java*** tiene muchas posibilidades: ejecución como aplicación independiente (***Stand-alone Application***), ejecución como ***applet***, ejecución como ***servlet***, etc. Un ***applet*** es una aplicación especial que se ejecuta dentro de un navegador o browser (por ejemplo *Netscape Navigator* o *Internet Explorer*) al cargar una página HTML desde un servidor ***Web***. El ***applet*** se descarga desde el servidor y no requiere instalación en el ordenador donde se encuentra el browser. Un ***servlet*** es una aplicación sin interface gráfica que se ejecuta en un servidor de Internet. La ejecución como aplicación independiente es análoga a los programas desarrollados con otros lenguajes.

Además de incorporar la ejecución como ***Applet***, ***Java*** permite fácilmente el desarrollo tanto de arquitecturas cliente-servidor como de aplicaciones distribuidas, consistentes en crear aplicaciones capaces de conectarse a otros ordenadores y ejecutar tareas en varios ordenadores simultáneamente, repartiendo por lo tanto el trabajo. Aunque también otros lenguajes de programación permiten crear aplicaciones de este tipo, ***Java*** incorpora en su propio ***API*** estas funcionalidades.

1.2 EL ENTORNO DE DESARROLLO DE JAVA

Existen distintos programas comerciales que permiten desarrollar código ***Java***. La compañía ***Sun***, creadora de ***Java***, distribuye gratuitamente el *Java(tm) Development Kit* (***JDK***). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en ***Java***. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado ***Debugger***). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la ***detección y corrección de errores***. Existe también una versión reducida del ***JDK***, denominada ***JRE*** (*Java Runtime Environment*) destinada únicamente a ejecutar código ***Java*** (no permite compilar).

Los ***IDE***s (*Integrated Development Environment*), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código ***Java***, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar ***Debug*** gráficamente, frente a la versión que incorpora el ***JDK*** basada en la utilización de una consola (denominada habitualmente ventana de comandos de MS-DOS, en ***Windows NT/95/98***) bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones

e forma mucho más rápida, incorporando en muchos casos librerías con ***componentes*** ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y ficheros resultantes de mayor tamaño que los basados en clases estándar.

1.2.1 El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el ***JDK***. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de ***Java*** (con extensión ****.java***). Si no encuentra errores en el código genera los ficheros compilados (con extensión ****.class***). En otro caso muestra la línea o líneas erróneas. En el ***JDK*** de ***Sun*** dicho compilador se llama ***javac.exe***. Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del ***JDK*** utilizada para obtener una información detallada de las distintas posibilidades.

1.2.2 La Java Virtual Machine

Tal y como se ha comentado al comienzo del capítulo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de ***Sun*** a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se planteó la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una "*máquina hipotética o virtual*", denominada ***Java Virtual Machine*** (***JVM***). Es esta ***JVM*** quien ***interpreta*** este código neutro convirtiéndolo a código particular de la CPU utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La ***JVM*** es el intérprete de ***Java***. Ejecuta los "***bytecodes***" (ficheros compilados con extensión ****.class***) creados por el compilador de ***Java*** (***javac.exe***). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado ***JIT*** (*Just-In-Time Compiler*), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

1.2.3 Las variables PATH y CLASSPATH

El desarrollo y ejecución de aplicaciones en ***Java*** exige que las herramientas para compilar (***javac.exe***) y ejecutar (***java.exe***) se encuentren accesibles. El ordenador, desde una ventana de comandos de MS-DOS, sólo es capaz de ejecutar los programas que se encuentran en los directorios indicados en la variable ***PATH*** del ordenador (o en el directorio activo). Si se desea compilar o ejecutar código en ***Java***, el directorio donde se encuentran estos programas (***java.exe*** y ***javac.exe***) deberá encontrarse en el ***PATH***. Tecleando ***PATH*** en una ventana de comandos de MS-DOS se muestran los nombres de directorios incluidos en dicha variable de entorno.

Java utiliza además una nueva variable de entorno denominada ***CLASSPATH***, la cual determina dónde buscar tanto las clases o librerías de ***Java*** (el ***API*** de ***Java***) como otras clases de usuario. A partir de la versión 1.1.4 del JDK no es necesario indicar esta variable, salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho ***JDK***. La variable ***CLASSPATH*** puede incluir la ruta de directorios o ficheros ****.zip*** o ****.jar*** en los que se encuentren los ficheros ****.class***. En el caso de los ficheros ****.zip*** hay que observar que los ficheros en él incluidos

no deben estar comprimidos. En el caso de archivos ****.jar*** existe una herramienta (***jar.exe***), incorporada en el ***JDK***, que permite generar estos ficheros a partir de los archivos compilados ****.class***. Los ficheros ****.jar*** son archivos comprimidos y por lo tanto ocupan menos espacio que los archivos ****.class*** por separado o que el fichero ****.zip*** equivalente.

Una forma general de indicar estas dos variables es crear un fichero ***batch*** de MS-DOS (****.bat***) donde se indiquen los valores de dichas variables. Cada vez que se abra una ventana de MS-DOS será necesario ejecutar este fichero ****.bat*** para asignar adecuadamente estos valores. Un posible fichero llamado ***jdk117.bat***, podría ser como sigue:

```
set JAVAPATH=C:\jdk1.1.7
set PATH=.;%JAVAPATH%\bin;%PATH%
set CLASSPATH=.\;%JAVAPATH%\lib\classes.zip;%CLASSPATH%
```

lo cual sería válido en el caso de que el ***JDK*** estuviera situado en el directorio ***C:\jdk1.1.7***.

Si no se desea tener que ejecutar este fichero cada vez que se abre una consola de MS-DOS es necesario indicar estos cambios de forma "permanente". La forma de hacerlo difiere ***entre Windows 95/98*** y ***Windows NT***. En ***Windows 95/98*** es necesario modificar el fichero ***Autoexec.bat*** situado en C:\, añadiendo las líneas antes mencionadas. Una vez rearrancado el ordenador estarán presentes en cualquier consola de MS-DOS que se cree. La modificación al fichero ***Autoexec.bat*** en ***Windows 95/98*** será la siguiente:

```
set JAVAPATH=C:\jdk1.1.7
set PATH=.;%JAVAPATH%\bin;%PATH%
set CLASSPATH=
```

donde en la tercera línea debe incluir la ruta de los ficheros donde están las clases de ***Java***. En el caso de utilizar **Windows NT** se añadirá la variable **PATH** en el cuadro de diálogo que se abre con *Start -> Settings -> Control Panel -> System -> Environment -> User Variables for NombreUsuario*.

También es posible utilizar la opción ***–classpath*** en el momento de llamar al compilador ***javac.exe*** o al intérprete ***java.exe***. En este caso los ficheros ****.jar*** deben ponerse con el nombre completo en el ***CLASSPATH***: no basta poner el ***PATH*** o directorio en el que se encuentra. Por ejemplo, si se desea compilar y ejecutar el fichero ***ContieneMain.java***, y éste necesitara la librería de clases ***G:\MyProject\OtherClasses.jar***, además de las incluidas en el ***CLASSPATH***, la forma de compilar y ejecutar sería:

```
javac -classpath .\;G:\MyProject\OtherClasses.jar ContieneMain.java
java -classpath .\;G:\MyProject\OtherClasses.jar ContieneMain
```

Se aconseja consultar la ayuda correspondiente a la versión que se esté utilizando, debido a que existen pequeñas variaciones entre las distintas versiones del JDK.

Cuando un fichero ***filename.java*** se compila y en ese directorio existe ya un fichero ***filename.class***, se comparan las fechas de los dos ficheros. Si el fichero ***filename.java*** es más antiguo que el ***filename.class*** no se produce un nuevo fichero ***filename.class***. Esto sólo es válido para ficheros ****.class*** que se corresponden con una clase ***public***.

1.3 UN EJEMPLO COMPLETO COMENTADO

Este ejemplo contiene algunas de las características más importantes de ***Java***: *clases, herencia, interfaces, gráficos, polimorfismo*, etc. Las sentencias se numeran en cada fichero, de modo que

resulta más fácil hacer referencia a ellas en los comentarios. La ejecución de este programa imprime algunas líneas en la consola MS-DOS y conduce a crear la ventana mostrada en la Figura 1.1.

1.3.1 Clase Ejemplo1

A continuación se muestra el programa principal, contenido en el fichero ***Ejemplo1.java***. En realidad, este programa principal lo único que hace es utilizar la clase ***Geometría*** y sus clases derivadas. Es pues un programa puramente "usuario", a pesar de lo cual hay que definirlo dentro de una clase, como todos los programas en ***Java***.

```
1.    // fichero Ejemplo1.java
2.    import java.util.Vector;
3.    import java.awt.*;

4.    class Ejemplo1 {
5.        public static void main(String arg[]) throws InterruptedException
6.        {
7.            System.out.println("Comienza main()...");
8.            Circulo c = new Circulo(2.0, 2.0, 4.0);
9.            System.out.println("Radio = " + c.r + " unidades.");
10.           System.out.println("Centro = (" + c.x + "," + c.y + ") unidades.");
11.           Circulo c1 = new Circulo(1.0, 1.0, 2.0);
12.           Circulo c2 = new Circulo(0.0, 0.0, 3.0);
13.           c = c1.elMayor(c2);
14.           System.out.println("El mayor radio es " + c.r + ".");
15.           c = new Circulo();   //  c.r = 0.0;
16.           c = Circulo.elMayor(c1, c2);
17.           System.out.println("El mayor radio es " + c.r + ".");

18.           VentanaCerrable ventana =
19.               new VentanaCerrable("Ventana abierta al mundo...");
20.           ArrayList v = new ArrayList();

21.           CirculoGrafico cg1 = new CirculoGrafico(200, 200, 100, Color.red);
22.           CirculoGrafico cg2 = new CirculoGrafico(300, 200, 100, Color.blue);
23.           RectanguloGrafico rg = new
24.               RectanguloGrafico(50, 50, 450, 350, Color.green);

25.           v.add(cg1);
26.           v.add(cg2);
27.           v.add(rg);

28.           PanelDibujo mipanel = new PanelDibujo(v);
29.           ventana.add(mipanel);
30.           ventana.setSize(500, 400);
31.           ventana.setVisible(true);
32.           System.out.println("Termina main()...");

33.       } // fin de main()

34.   } // fin de class Ejemplo1
```

La sentencia 1 es simplemente un comentario que contiene el nombre del fichero. El compilador de ***Java*** ignora todo lo que va desde los caracteres // hasta el final de la línea.

Las sentencias 2 y 3 "importan" clases de los ***packages*** de ***Java***, esto es, hacen posible acceder a dichas clases utilizando nombres cortos. Por ejemplo, se puede acceder a la clase ***Vector*** simplemente con el nombre ***Vector*** en lugar de con el nombre completo ***java.util.Vector***, por haber introducido la sentencia ***import*** de la línea 2. Un ***package*** es una agrupación de clases que tienen

una finalidad relacionada. Existe una jerarquía de ***packages*** que se refleja en nombres compuestos, separados por un punto (.). Es habitual nombrar los ***packages*** con letras minúsculas (como ***java.util*** o ***java.awt***), mientras que los nombres de las clases suelen empezar siempre por una letra mayúscula (como ***Vector***). El asterisco (*) de la sentencia 3 indica que se importan todas las clases del ***package***. Hay un ***package***, llamado ***java.lang***, que se importa siempre automáticamente. Las clases de ***java.lang*** se pueden utilizar directamente, sin importar el ***package***.

La sentencia 4 indica que se comienza a definir la clase ***Ejemplo1***. La definición de dicha clase va entre ***llaves*** {}. Como también hay otras construcciones que van entre llaves, es habitual indentar o sangrar el código, de forma que quede claro donde empieza (línea 4) y donde termina (línea 34) la definición de la clase. En ***Java*** todo son clases: no se puede definir una variable o una función que no pertenezca a una clase. En este caso, la clase ***Ejemplo1*** tiene como única finalidad acoger al método ***main()***, que es el programa principal del ejemplo. Las clases utilizadas por ***main()*** son mucho más importantes que la propia clase ***Ejemplo1***. Se puede adelantar ya que una ***clase*** es una agrupación de ***variables miembro*** (*datos*) y ***funciones miembro*** (*métodos*) que operan sobre dichos datos y permiten comunicarse con otras clases. Las clases son verdaderos ***tipos de variables*** o datos, creados por el usuario. Un ***objeto*** (en ocasiones también llamado ***instancia***) es una variable concreta de una clase, con su propia copia de las variables miembro.

Las líneas 5-33 contienen la definición del programa principal de la aplicación, que en ***Java*** siempre se llama ***main()***. La ejecución siempre comienza por el programa o método ***main()***. La palabra ***public*** indica que esta función puede ser utilizada por cualquier clase; la palabra ***static*** indica que es un ***método de clase***, es decir, un método que puede ser utilizado aunque no se haya creado ningún objeto de la clase ***Ejemplo1*** (que de hecho, no se han creado); la palabra ***void*** indica que este método no tiene valor de retorno. A continuación del nombre aparecen, entre paréntesis, los argumentos del método. En el caso de ***main()*** el argumento es siempre un *vector* o *array* (se sabe por la presencia de los corchetes []), en este caso llamado ***arg***, de cadenas de caracteres (objetos de la clase ***String***). Estos argumentos suelen ser parámetros que se pasan al programa en el momento de comenzar la ejecución (por ejemplo, el nombre del fichero donde están los datos).

El ***cuerpo*** (*body*) del método ***main()***, definido en las líneas 6-33, va también encerrado entre ***llaves*** {...}. A un conjunto de sentencias encerrado entre llaves se le suele llamar ***bloque***. Es conveniente *indentar* para saber dónde empieza y dónde terminan los bloques del método ***main()*** y de la clase ***Ejemplo1***. Los bloques nunca pueden estar entrecruzados; un bloque puede contener a otro, pero nunca se puede cerrar el bloque exterior antes de haber cerrado el interior.

La sentencia 7 (`System.out.println("Comienza main()...");`) ***imprime*** una *cadena de caracteres* o *String* en la salida estándar del sistema, que normalmente será una ventana de MS-DOS o una ventana especial del entorno de programación que se utilice (por ejemplo ***Visual J++***, de ***Microsoft***). Para ello se utiliza el método ***println()***, que está asociado con una variable ***static*** llamada ***out***, perteneciente a la clase ***System*** (en el ***package*** por defecto, ***java.lang***). Una ***variable miembro static***, también llamada ***variable de clase***, es una variable miembro que es única para toda la clase y que existe aunque no se haya creado ningún objeto de la clase. La variable ***out*** es una variable ***static*** de la clase ***System***. La sentencia 7, al igual que las que siguen, termina con el carácter ***punto y coma*** (;).

La sentencia 8 (`Circulo c = new Circulo(2.0, 2.0, 4.0);`) es muy propia de ***Java***. En ella se crea un ***objeto*** de la clase ***Circulo***, que se define en el Apartado 1.3.4, en la página 11. Esta sentencia es equivalente a las dos sentencias siguientes:

```
Circulo c;
c = new Circulo(2.0, 2.0, 4.0);
```

que quizás son más fáciles de explicar. En primer lugar se crea una ***referencia*** llamada ***c*** a un objeto de la clase ***Circulo***. Crear una referencia es como crear un "nombre" válido para referirse a un objeto de la clase ***Circulo***. A continuación, con el operador ***new*** se crea el objeto propiamente dicho. Puede verse que el nombre de la clase va seguido por tres argumentos entre paréntesis. Estos argumentos se le pasan al ***constructor*** de la clase como datos concretos para crear el objeto (en este caso los argumentos son las dos coordenadas del centro y el radio).

Interesa ahora insistir un poco más en la diferencia entre ***clase*** y ***objeto***. La *clase* ***Circulo*** es lo genérico: es el patrón o modelo para crear círculos concretos. El *objeto* ***c*** es un círculo concreto, con su centro y su radio. De la clase ***Circulo*** se pueden crear tantos objetos como se desee; la clase dice que cada objeto necesita tres datos (las dos coordenadas del centro y el radio) que son las ***variables miembro*** de la clase. Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores, distintos de los demás objetos de la clase.

La sentencia 9 (`System.out.println("Radio = " + c.r + " unidades.");`) imprime por la salida estándar una cadena de texto que contiene el valor del radio. Esta cadena de texto se compone de tres sub-cadenas, unidas mediante el ***operador de concatenación*** (+). Obsérvese cómo se accede al radio del objeto ***c***: el nombre del objeto seguido del nombre de la variable miembro ***r***, unidos por el operador punto (***c.r***). El valor numérico del radio se convierte automáticamente en cadena de caracteres. La sentencia 10 es similar a la 9, imprimiendo las coordenadas del centro del círculo.

Las sentencias 11 y 12 crean dos nuevos objetos de la clase ***Circulo***, llamados ***c1*** y ***c2***.

La sentencia 13 (`c = c1.elMayor(c2);`) utiliza el método ***elMayor()*** de la clase ***Circulo***. Este método compara los radios de dos círculos y devuelve como ***valor de retorno*** una ***referencia*** al círculo que tenga mayor radio. Esa referencia se almacena en la referencia previamente creada ***c***. Un punto importante es que todos los métodos de ***Java*** (excepto los *métodos de clase* o *static*) se aplican a un objeto de la clase por medio del operador punto (por ejemplo, `c1.elMayor()`). El otro objeto (***c2***) se pasa como argumento entre paréntesis. Obsérvese la forma "asimétrica" en la que se pasan los dos argumentos al método ***elMayor()***. De ordinario se llama ***argumento implícito*** a ***c1***, mientras que ***c2*** sería el ***argumento explícito*** del método.

La sentencia 14 imprime el resultado de la comparación anterior y la sentencia 15 crea un nuevo objeto de la clase ***Circulo*** guardándolo en la referencia ***c***. En este caso no se pasan argumentos al constructor de la clase. Eso quiere decir que deberá utilizar algunos valores "por defecto" para el centro y el radio. Esta sentencia anula o borra el resultado de la primera comparación de radios, de modo que se pueda comprobar el resultado de la segunda comparación.

La sentencia 16 (`c = Circulo.elMayor(c1, c2);`) vuelve a utilizar un método llamado ***elMayor()*** para comparar dos círculos: ¿Se trata del mismo método de la sentencia 13, utilizado de otra forma? No. Se trata de un método diferente, aunque tenga el mismo nombre. A las funciones o métodos que son diferentes porque tienen distinto código, aunque tengan el mismo nombre, se les llama ***funciones sobrecargadas*** (*overloaded*). Las funciones sobrecargadas se diferencian por el numero y tipo de sus argumentos. El método de la sentencia 13 tiene un único argumento, mientras que el de la sentencia 16 tiene dos (en todos los casos objetos de la clase ***Circulo***). En realidad, el método de la sentencia 16 es un ***método static*** (o ***método de clase***), esto es, un método que no

necesita ningún objeto como argumento implícito. Los métodos ***static*** suelen ir *precedidos por el nombre de la clase* y el operador punto (***Java*** también permite que vayan precedidos por el nombre de cualquier objeto, pero es considerada una nomenclatura más confusa.). La sentencia 16 es absolutamente equivalente a la sentencia 13, pero el método ***static*** de la sentencia 16 es más "simétrico". Las sentencias 17 y 18 no requieren ya comentarios especiales.

Las sentencias 18-31 tienen que ver con la parte gráfica del ejemplo. En las líneas 18-19 (`VentanaCerrable ventana = new VentanaCerrable("Ventana abierta al mundo...");`) se crea una ventana para dibujar sobre ella. Una ventana es un objeto de la clase ***Frame***, del package ***java.awt***. La clase ***VentanaCerrable***, explicada en el Apartado 1.3.9 en la página 18, añade a la clase ***Frame*** la capacidad de responder a los ***eventos*** que provocan el cierre de una ventana. La cadena que se le pasa como argumento es el título que aparecerá en la ventana (ver Figura 1.1). En la sentencia 20 (`Vector v = new Vector();`) se crea un objeto de la clase ***ArrayList*** (contenida o definida en el package ***java.util***). La clase ***ArrayList*** permite almacenar referencias a objetos de distintas clases. En este caso se utilizará para almacenar referencias a varias figuras geométricas diferentes.

Las siguientes sentencias 21-27 crean elementos gráficos y los incluyen en la lista ***v*** para ser dibujados más tarde en el objeto de la clase ***PanelDibujo***. Los objetos de la clase ***Circulo*** creados anteriormente no eran objetos aptos para ser dibujados, pues sólo tenían información del centro y el radio, y no del color de línea. Las clases ***RectanguloGrafico*** y ***CirculoGrafico***, definidas en los Apartados 1.3.6 y 1.3.7, derivan respectivamente de las clases ***Rectangulo*** (Apartado 1.3.3) y ***Circulo*** (Apartado 1.3.4), heredando de dichas clases sus variables miembro y métodos, añadiendo la información y los métodos necesarios para poder dibujarlos en la pantalla. En las sentencias 21-22 se definen dos objetos de la clase ***CirculoGrafico***; a las coordenadas del centro y al radio se une el color de la línea. En la sentencia 23-24 se define un objeto de la clase ***RectanguloGrafico***, especificando asimismo un color, además de las coordenadas del vértice superior izquierdo, y del vértice inferior derecho. En las sentencias 25-27 los objetos gráficos creados se añaden al vector ***v***, utilizando el método ***addElement()*** de la propia clase ***Vector***.

En la sentencia 28 (`PanelDibujo mipanel = new PanelDibujo(v);`) se crea un objeto de la clase ***PanelDibujo***, definida en el Apartado 1.3.8. Por decirlo de alguna manera, los objetos de dicha clase son ***paneles***, esto es *superficies en las que se puede dibujar*. Al constructor de ***PanelDibujo*** se le pasa como argumento el vector ***v*** con las referencias a los objetos a dibujar. La sentencia 29 (`ventana.add(mipanel);`) añade o incluye el ***panel*** (la superficie de dibujo) en la ventana; la sentencia 30 (`ventana.setSize(500, 400);`) establece el tamaño de la ventana en *pixels*; finalmente, la sentencia 31 (`ventana.setVisible(true);`) hace visible la ventana creada.

¿Cómo se consigue que se dibuje todo esto? La clave está en la serie de órdenes que se han ido dando al computador. La clase ***PanelDibujo*** deriva de la clase ***Container*** a través de ***Panel***, y redefine el método ***paint()*** de ***Container***. En este método, explicado en el Apartado 1.3.8, se realiza el dibujo de los objetos gráficos creados. El usuario no tiene que preocuparse de llamar al método ***paint()***, pues se llama de modo automático cada vez que el sistema operativo tiene alguna razón para ello (por ejemplo cuando se crea la ventana, cuando se mueve, cuando se minimiza o maximiza, cuando aparece después de haber estado oculta, etc.). La Figura 1.1 muestra la ventana resultante de la ejecución del programa ***main()*** de la clase ***Ejemplo1***. Para entender más a fondo este resultado es necesario considerar detenidamente las clases definidas en los apartados que siguen.

1.3.2 Clase Geometria

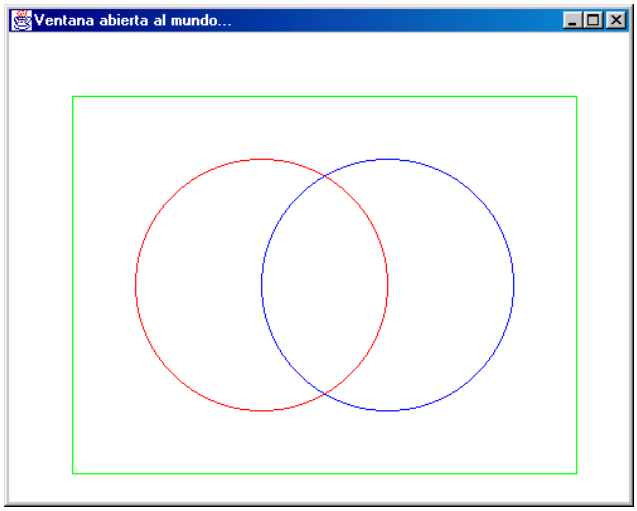


Figura 1.1. Resultado de la ejecución del Ejemplo1.

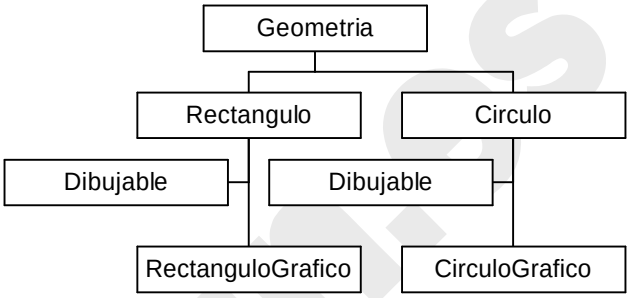


Figura 1.2. Jerarquía de clases utilizadas.

En este apartado se describe la clase más importante de esta aplicación. Es la más importante no en el sentido de lo que hace, sino en el de que las demás clases "derivan" de ella, o por decirlo de otra forma, se apoyan o cuelgan de ella. La Figura 1.2 muestra la jerarquía de clases utilizada en este ejemplo. La clase ***Geometria*** es la base de la jerarquía. En realidad no es la base, pues en ***Java*** la clase base es siempre la clase ***Object***. Siempre que no se diga explícitamente que una clase deriva de otra, deriva implícitamente de la clase ***Object*** (definida en el package ***java.lang***). De las clases ***Rectangulo*** y ***Circulo*** derivan respectivamente las clases ***RectanguloGrafico*** y ***CirculoGrafico***. En ambos casos está por en medio un elemento un poco especial donde aparece la palabra ***Dibujable***. En términos de ***Java***, ***Dibujable*** es una ***interface***. Más adelante se verá qué es una ***interface***.

Se suele utilizar la nomenclatura de ***super-clase*** y ***sub-clase*** para referirse a la clase *padre* o *hija* de una clase determinada. Así ***Geometría*** es una ***super-clase*** de ***Circulo***, mientras que ***CirculoGrafico*** es una ***sub-clase***.

En este ejemplo sólo se van a dibujar rectángulos y círculos. De la clase ***Geometría*** van a derivar las clases ***Rectangulo*** y ***Circulo***. Estas clases tienen en común que son "geometrías", y como tales tendrán ciertas características comunes como un *perímetro* y un *área*. Un aspecto importante a considerar es que no va a haber nunca objetos de la clase ***Geometria***, es decir "geometrías a secas". Una clase de la que no va a haber objetos es una ***clase abstracta***, y como tal puede ser declarada. A continuación se muestra el fichero ***Geometria.java*** en el que se define dicha clase:

```
1.    // fichero Geometria.java

2.    public abstract class Geometria {
3.        // clase abstracta que no puede tener objetos

4.        public abstract double perimetro();
5.        public abstract double area();

6.    }
```

La clase ***Geometria*** se declara como ***public*** para permitir que sea utilizada por cualquier otra clase, y como ***abstract*** para indicar que no se permite crear objetos de esta clase. Es característico de las clases tener variables y funciones miembro. La clase ***Geometria*** no define ninguna variable miembro, pero sí ***declara*** dos métodos: ***perímetro()*** y ***area()***. Ambos métodos se declaran como ***public*** para que puedan ser llamados por otras clases y como ***abstract*** para indicar que no se da ninguna ***definición*** -es decir ningún código- para ellos. Interesa entender la diferencia entre ***declaración*** (la primera línea o header del método) y ***definición*** (todo el código del método, incluyendo la primera línea). Se indica también que su ***valor de retorno*** -el resultado- va a ser un ***double*** y que no tienen argumentos (obtendrán sus datos a partir del objeto que se les pase como argumento implícito). Es completamente lógico que no se definan en esta clase los métodos ***perimetro()*** y ***area()***: la forma de calcular un perímetro o un área es completamente distinta en un rectángulo y en un círculo, y por tanto estos métodos habrá que definirlos en las clases ***Rectangulo*** y ***Circulo***. En la clase ***Geometria*** lo único que se puede decir es cómo serán dichos métodos, es decir su nombre, el número y tipo de sus argumentos y el tipo de su valor de retorno.

1.3.3 Clase Rectangulo

Según el diagrama de clases de la Figura 1.2 la clase ***Rectangulo*** deriva de ***Geometria***. Esto se indica en la sentencia 2 con la palabra ***extends*** (en negrita en el listado de la clase).

```
1.    // fichero Rectangulo.java

2.    public class Rectangulo extends Geometria {
3.        // definición de variables miembro de la claes
4.        private static int numRectangulos = 0;
5.        protected double x1, y1, x2, y2;

6.        // constructores de la clase
7.        public Rectangulo(double p1x, double p1y, double p2x, double p2y) {
8.            x1 = p1x;
9.            x2 = p2x;
10.           y1 = p1y;
11.           y2 = p2y;
12.           numRectangulos++;
13.       }
14.       public Rectangulo(){ this(0, 0, 1.0, 1.0); }

15.       // definición de métodos
16.       public double perimetro() { return 2.0 * ((x1-x2)+(y1-y2)); }
17.       public double area() { return (x1-x2)*(y1-y2); }

18.   } // fin de la clase Rectangulo
```

La clase ***Rectangulo*** define cinco variables miembro. En la sentencia 4 (`private static int numRectangulos = 0;`) se define una variable miembro ***static***. Las variables miembro ***static*** se caracterizan por ser propias de la clase y no de cada objeto. En efecto, la variable ***numRectangulos*** pretende llevar cuenta en todo momento del número de objetos de esta clase que se han creado. No tiene sentido ni sería práctico en absoluto que cada objeto tuviera su propia copia de esta variable, teniendo además que actualizarla cada vez que se crea o se destruye un nuevo rectángulo. De la variable ***numRectangulos***, que en la sentencia 4 se inicializa a cero, se mantiene una única copia para toda la clase. Además esta variable es *privada* (***private***), lo cual quiere decir que sólo las funciones miembro de esta clase tienen permiso para utilizarla.

La sentencia 5 (`protected double x1, y1, x2, y2;`) define cuatro nuevas variables miembro, que representan las coordenadas de dos vértices opuestos del rectángulo. Las cuatro son de tipo

double. El declararlas como ***protected*** indica que sólo esta clase, las clases que deriven de ella y las clases del propio package tienen permiso para utilizarlas.

Las sentencias 7-14 definen los ***constructores*** de la clase. Los constructores son unos métodos o funciones miembro muy importantes. Como se puede ver, no tienen ***valor de retorno*** (ni siquiera ***void***) y ***su nombre coincide con el de la clase***. Los constructores son un ejemplo típico de ***métodos sobrecargados*** (*overloaded*): en este caso hay dos constructores, el segundo de los cuales no tiene ningún argumento, por lo que se llama ***constructor por defecto***. Las sentencias 7-13 definen el ***constructor general***. Este constructor recibe cuatro argumentos con cuatro valores que asigna a las cuatro variables miembro. La sentencia 12 incrementa en una unidad (esto es lo que hace el operador ++, típico de C y C++, de los que ***Java*** lo ha heredado) el número de rectángulos creados hasta el momento.

La sentencia 14 (`public Rectangulo(){ this(0, 0, 1.0, 1.0); }`) define un segundo constructor, que por no necesitar argumentos es un ***constructor por defecto***. ¿Qué se puede hacer cuando hay que crear un rectángulo sin ningún dato? Pues algo realmente sencillo: en este caso se ha optado por crear un rectángulo de lado unidad cuyo primer vértice coincide con el origen de coordenadas. Obsérvese que este constructor en realidad no tiene código para inicializar las variables miembro, limitándose a llamar al constructor general previamente creado, utilizando para ello la palabra ***this*** seguida del valor por defecto de los argumentos. Ya se verá que la palabra ***this*** tiene otro uso aún más importante en ***Java***.

Las sentencias 16 (`public double perimetro() { return 2.0 * ((x1-x2)+(y1-y2)); }`) y 17 (`public double area() { return (x1-x2)*(y1-y2); }`) contienen la definición de los métodos miembro ***perimetro()*** y ***area()***. La declaración coincide con la de la clase ***Geometría***, pero aquí va seguida del cuerpo del método entre llaves {...}. Las fórmulas utilizadas son las propias de un rectángulo.

1.3.4 Clase Circulo

A continuación se presenta la definición de la clase ***Circulo***, también derivada de ***Geometria***, y que resulta bastante similar en muchos aspectos a la clase ***Rectangulo***. Por eso, en este caso las explicaciones serán un poco más breves, excepto cuando aparezcan cosas nuevas.

```
1.    // fichero Circulo.java

2.    public class Circulo extends Geometria {
3.        static int numCirculos = 0;
4.        public static final double PI=3.14159265358979323846;
5.        public double x, y, r;

6.        public Circulo(double x, double y, double r) {
7.            this.x=x; this.y=y; this.r=r;
8.            numCirculos++;
9.        }

10.       public Circulo(double r) { this(0.0, 0.0, r); }
11.       public Circulo(Circulo c) { this(c.x, c.y, c.r); }
12.       public Circulo() { this(0.0, 0.0, 1.0); }

13.       public double perimetro() { return 2.0 * PI * r; }
14.       public double area() { return PI * r * r; }

15.       // método de objeto para comparar círculos
```

```
16.         public Circulo elMayor(Circulo c) {
17.             if (this.r>=c.r) return this; else return c;
18.         }

19.         // método de clase para comparar círculos
20.         public static Circulo elMayor(Circulo c, Circulo d) {
21.             if (c.r>=d.r) return c; else return d;
22.         }

23.     } // fin de la clase Circulo
```

La sentencia 3 (`static int numCirculos = 0;`) define una variable ***static*** o ***de clase*** análoga a la de la clase ***Rectangulo***. En este caso no se ha definido como ***private***. Cuando no se especifican permisos de acceso (***public***, ***private*** o ***protected***) se supone la opción por defecto, que es ***package***. Con esta opción la variable o método correspondiente puede ser utilizada por todas las clases del ***package*** y sólo por ellas. Como en este ejemplo no se ha definido ningún ***package***, se utiliza el ***package por defecto*** que es el directorio donde están definidas las clases. Así pues, la variable ***numCirculos*** podrá ser utilizada sólo por las clases que estén en el mismo directorio que ***Circulo***.

La sentencia 4 (`public static final double PI=3.14159265358979323846;`) define también una variable ***static***, pero contiene una palabra nueva: ***final***. Una variable ***final*** tiene como característica el que su valor no puede ser modificado, o lo que es lo mismo, es una ***constante***. Es muy lógico definir el número p como constante, y también es razonable que sea una constante ***static*** de la clase ***Circulo***, de forma que sea compartida por todos los métodos y objetos que se creen. La sentencia 5 (`public double x, y, r;`) define las ***variables miembro de objeto***, que son las coordenadas del centro y el radio del círculo.

La sentencia 6-9 define el constructor general de la clase ***Circulo***. En este caso tiene una peculiaridad y es que el nombre de los argumentos (***x***, ***y***, ***r***) coincide con el nombre de las variables miembro. Esto es un problema, porque como se verá más adelante ***los argumentos de un método son variables locales*** que sólo son visibles dentro del bloque {...} del método, que se destruyen al salir del bloque y que *ocultan otras variables* de ámbito más general que tengan esos mismos nombres. En otras palabras, si en el código del constructor se utilizan las variables (***x***, ***y***, ***r***) se está haciendo referencia a los argumentos del método y no a las variables miembro. La sentencia 7 indica cómo se resuelve este problema. Para cualquier método ***no static*** de una clase, la palabra ***this*** es una referencia al objeto -el ***argumento implícito***- sobre el que se está aplicando el método. De esta forma, ***this.x*** se refiere a la variable miembro, mientras que ***x*** es el argumento del constructor.

Las sentencias 10-12 representan otros tres constructores de la clase (métodos sobrecargados), que se diferencian en el número y tipo de argumentos. Los tres tienen en común el realizar su papel llamando al constructor general previamente definido, al que se hace referencia con la palabra ***this*** (en este caso el significado de ***this*** no es exactamente el del argumento implícito). Al constructor de la sentencia 10 sólo se le pasa el radio, con lo cual construye un círculo con ese radio centrado en el origen de coordenadas. Al constructor de la sentencia 11 se le pasa otro objeto de la clase ***Circulo***, del cual saca una copia. El constructor de la sentencia 12 es un ***constructor por defecto***, al que no se le pasa ningún argumento, que crea un círculo de radio unidad centrado en el origen.

Las sentencias 13 y 14 definen los métodos ***perimetro()*** y ***area()***, declarados como ***abstract*** en la clase ***Geometria***, de modo adecuado para los círculos.

Las sentencias 16-18 definen ***elMayor()***, que es un ***método de objeto*** para comparar círculos. Uno de los círculos le llega como argumento implícito y el otro como argumento explícito. En la

sentencia 17 se ve cómo al radio del argumento implícito se accede en la forma ***this.r*** (se podría acceder también simplemente con ***r***, pues no hay ninguna variable local que la oculte), y al del argumento explícito como ***c.r***, donde ***c*** es el nombre del objeto pasado como argumento. La sentencia ***return*** devuelve una referencia al objeto cuyo radio sea mayor. Cuando éste es el argumento implícito se devuelve ***this***.

Las sentencias 20-22 presentan la definición de otro método ***elMayor()***, que en este caso es un método de clase (definido como ***static***), y por tanto no tiene argumento implícito. Los dos objetos a comparar se deben pasar como argumentos explícitos, lo que hace el código muy fácil de entender. Es importante considerar que en ambos casos lo que se devuelve como valor de retorno no es el objeto que constituye el mayor círculo, sino una ***referencia*** (un *nombre*, por decirlo de otra forma).

1.3.5 Interface Dibujable

El diagrama de clases de la Figura 1.2 indica que las clases ***RectanguloGrafico*** y ***CirculoGrafico*** son el resultado, tanto de las clases ***Rectangulo*** y ***Circulo*** de las que derivan, como de la ***interface Dibujable***, que de alguna manera interviene en el proceso.

El concepto de ***interface*** es muy importante en ***Java***. A diferencia de C++, ***Java*** no permite ***herencia múltiple***, esto es, no permite que una clase derive de dos clases distintas heredando de ambas métodos y variables miembro. La herencia múltiple es fuente de problemas, pero en muchas ocasiones es una característica muy conveniente. Las ***interfaces*** de ***Java*** constituyen una alternativa a la herencia múltiple con importantes ventajas prácticas y de "estilo de programación".

Una ***interface*** es un conjunto de ***declaraciones de métodos*** (sin implementación, es decir, sin definir el código de dichos métodos). La declaración consta del tipo del valor de retorno y del nombre del método, seguido por el tipo de los argumentos entre paréntesis. Cuando una clase implementa una determinada ***interface***, se compromete a dar una ***definición*** a todos los métodos de la ***interface***. En cierta forma una ***interface*** se parece a una clase ***abstract*** cuyos métodos son todos ***abstract***. La ventaja de las ***interfaces*** es que no están sometidas a las más rígidas normas de las clases; por ejemplo, una clase no puede heredar de dos clases ***abstract***, pero sí puede implementar varias ***interfaces***.

Una de las ventajas de las ***interfaces*** de ***Java*** es el establecer pautas o *modos de funcionamiento* similares para clases que pueden estar o no relacionadas mediante herencia. En efecto, todas las clases que implementan una determinada ***interface*** soportan los métodos declarados en la ***interface*** y en este sentido se comportan de modo similar. Las ***interfaces*** pueden también relacionarse mediante mecanismos de ***herencia***, con más flexibilidad que las ***clases***. Más adelante se volverá con más detenimiento sobre este tema, muy importante para muchos aspectos de ***Java***. El fichero ***Dibujable.java*** define la interface ***Dibujable***, mostrada a continuación.

```
1.    // fichero Dibujable.java

2.    import java.awt.Graphics;

3.    public interface Dibujable {
4.        public void setPosicion(double x, double y);
5.        public void dibujar(Graphics dw);
6.    }
```

La interface ***Dibujable*** está dirigida a incorporar, en las clases que la implementen, la capacidad de dibujar sus objetos. El listado muestra la declaración de los métodos ***setPosicion()*** y ***dibujar()***. La declaración de estos métodos no tiene nada de particular. Como el método ***dibujar()*** utiliza como argumento un objeto de la clase ***Graphics***, es necesario importar dicha clase. Lo importante es que si las clases ***RectanguloGrafico*** y ***CirculoGrafico*** implementan la interface ***Dibujable*** sus objetos podrán ser representados gráficamente en pantalla.

1.3.6 Clase RectanguloGrafico

La clase ***RectanguloGrafico*** deriva de ***Rectangulo*** (lo cual quiere decir que hereda sus métodos y variables miembro) e implementa la interface ***Dibujable*** (lo cual quiere decir que ***debe definir*** los métodos declarados en dicha interface). A continuación se incluye la definición de dicha clase.

```
1.    // Fichero RectanguloGrafico.java

2.    import java.awt.Graphics;
3.    import java.awt.Color;

4.    class RectanguloGrafico extends Rectangulo implements Dibujable {
5.        // nueva variable miembro
6.        Color color;

7.        // constructor
8.        public RectanguloGrafico(double x1, double y1, double x2, double y2,
9.                                 Color unColor) {
10.           // llamada al constructor de Rectangulo
11.           super(x1, y1, x2, y2);
12.           this.color = unColor;     // en este caso this es opcional
13.       }

14.       // métodos de la interface Dibujable
15.       public void dibujar(Graphics dw) {
16.           dw.setColor(color);
17.           dw.drawRect((int)x1, (int)y1, (int)(x2-x1), (int)(y2-y1));
18.       }

19.       public void setPosicion(double x, double y) {
20.           ; // método vacío, pero necesario de definir
21.       }

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

Las sentencias 2 y 3 importan dos clases del package ***java.awt***. Otra posibilidad sería importar todas las clases de dicho ***package*** con la sentencia (`import java.awt.*;`).

La sentencia 4 indica que ***RectanguloGrafico*** deriva de la clase ***Rectangulo*** e implementa la interface ***Dibujable***. Recuérdese que mientras que sólo se puede derivar de una clase, se pueden implementar varias interfaces, en cuyo caso se ponen en el encabezamiento de la clase separadas por comas. La sentencia 6 (`Color color;`) define una nueva variable miembro que se suma a las que ya se tienen por herencia. Esta nueva variable es un objeto de la clase ***Color***.

Las sentencias 8-13 definen el constructor general de la clase, al cual le llegan los cinco argumentos necesarios para dar valor a todas las variables miembro. En este caso los nombres de los argumentos también coinciden con los de las variables miembro, pero sólo se utilizan para pasárselos al constructor de la super-clase. En efecto, la sentencia 11 (`super(x1, y1, x2, y2);`) contiene una novedad: para dar valor a las variables heredadas lo más cómodo es llamar al constructor de la clase padre o ***super-clase***, al cual se hace referencia con la palabra ***super***.

Las sentencias 14-18 y 19-21 definen los dos métodos declarados por la interface ***Dibujable***. El método ***dibujar()*** recibe como argumento un objeto ***dw*** de la clase ***Graphics***. Esta clase define un ***contexto*** para realizar operaciones gráficas en un panel, tales como el color de las líneas, el color de fondo, el tipo de letra a utilizar en los rótulos, etc. Más adelante se verá con más detenimiento este concepto. La sentencia 16 (`dw.setColor(color);`) hace uso un método de la clase ***Graphics*** para determinar el color con el que se dibujarán las líneas a partir de ese momento. La sentencia 17 (`dw.drawRect((int)x1, (int)y1, (int)(x2-x1), (int)(y2-y1));`) llama a otro método de esa misma clase que dibuja un rectángulo a partir de las coordenadas del vértice superior izquierdo, de la anchura y de la altura.

Java obliga a implementar o definir siempre todos los métodos declarados por la ***interface***, aunque no se vayan a utilizar. Esa es la razón de que las sentencias 19-21 definan un método vacío, que sólo contiene un carácter punto y coma. Como no se va a utilizar no importa que esté vacío, pero ***Java*** obliga a dar una definición o implementación.

1.3.7 Clase CirculoGrafico

A continuación se define la clase ***CirculoGrafico***, que deriva de la clase ***Circulo*** e implementa la interface ***Dibujable***. Esta clase es muy similar a la clase ***RectanguloGrafico*** y no requiere explicaciones especiales.

```
// fichero CirculoGrafico.java

import java.awt.Graphics;
import java.awt.Color;

public class CirculoGrafico extends Circulo implements Dibujable {
    // se heredan las variables y métodos de la clase Circulo

    Color color;

    // constructor
    public CirculoGrafico(double x, double y, double r, Color unColor) {
        // llamada al constructor de Circulo
        super(x, y, r);
        this.color = unColor;
    }

    // métodos de la interface Dibujable
    public void dibujar(Graphics dw) {
        dw.setColor(color);
        dw.drawOval((int)(x-r),(int)(y-r),(int)(2*r),(int)(2*r));

    }

    public void setPosicion(double x, double y) {
        ;
    }

} // fin de la clase CirculoGrafico
```

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 nombre sde 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 47.

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: ***Salida:***

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

```
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 145.

3. CLASES EN JAVA

Las ***clases*** son el centro de la ***Programación Orientada a Objetos*** (OOP - *Object Oriented Programming*). Algunos de los conceptos más importantes de la POO son los siguientes:

1. ***Encapsulación***. Las clases pueden ser declaradas como públicas (***public***) y como ***package*** (accesibles sólo para otras clases del ***package***). Las variables miembro y los métodos pueden ser ***public***, ***private***, ***protected*** y ***package***. De esta forma se puede controlar el acceso y evitar un uso inadecuado.
2. ***Herencia***. Una clase puede derivar de otra (***extends***), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede ***añadir*** nuevas variables y métodos y/o ***redefinir*** las variables y métodos heredados.
3. ***Polimorfismo***. Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo. Esto, como se ha visto en el ejemplo del Capítulo 1, facilita la programación y el mantenimiento del código.

En este Capítulo se presentan las ***clases*** y las ***interfaces*** tal como están implementadas en el lenguaje ***Java***.

3.1 CONCEPTOS BÁSICOS

3.1.1 Concepto de Clase

Una clase es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
[public] class Classname {
    // definición de variables y métodos
    ...
}
```

donde la palabra ***public*** es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del ***package***. Todos los métodos y variables deben ser definidos dentro del ***bloque*** {...} de la clase.

Un ***objeto*** (en inglés, ***instance***) es un ejemplar concreto de una clase. Las ***clases*** son como tipos de variables, mientras que los ***objetos*** son como variables concretas de un tipo determinado.

```
Classname unObjeto;
Classname otroObjeto;
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de ***Java*** deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (***extends***), hereda todas sus variables y métodos.
3. ***Java*** tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.

4. Una clase sólo puede heredar de una única clase (en ***Java*** no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de ***Object***. La clase ***Object*** es la base de toda la jerarquía de clases de ***Java***.
5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase ***public***. Este fichero se debe llamar como la clase ***public*** que contiene con extensión ****.java***. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
6. Si una clase contenida en un fichero no es ***public***, no es necesario que el fichero se llame como la clase.
7. Los métodos de una clase pueden referirse de modo global al ***objeto*** de esa clase al que se aplican por medio de la referencia ***this***.
8. Las clases se pueden agrupar en ***packages***, introduciendo una línea al comienzo del fichero (***package packageName;***). Esta agrupación en ***packages*** está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

3.1.2 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***. Las interfaces pueden definir también ***variables finales*** (constantes). Una ***clase*** puede implementar más de una ***interface***, representando una alternativa a la herencia múltiple.

En algunos aspectos los nombres de las ***interfaces*** pueden utilizarse en lugar de las ***clases***. Por ejemplo, las ***interfaces*** sirven para definir ***referencias*** a cualquier objeto de cualquiera de las clases que implementan esa ***interface***. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Éste es un aspecto importante del ***polimorfismo***.

Una ***interface*** puede derivar de otra o incluso de varias ***interfaces***, en cuyo caso incorpora las declaraciones de todos los métodos de las ***interfaces*** de las que deriva (a diferencia de las clases, las interfaces de ***Java*** sí tienen herencia múltiple).

3.2 EJEMPLO DE DEFINICIÓN DE UNA CLASE

A continuación se reproduce como ejemplo la clase ***Circulo***, explicada en el Apartado 1.3.4.

```
// fichero Circulo.java

public class Circulo extends Geometria {
    static int numCirculos = 0;
    public static final double PI=3.14159265358979323846;
    public double x, y, r;

    public Circulo(double x, double y, double r) {
        this.x=x; this.y=y; this.r=r;
        numCirculos++;
    }

    public Circulo(double r) { this(0.0, 0.0, r); }
    public Circulo(Circulo c) { this(c.x, c.y, c.r); }
    public Circulo() { this(0.0, 0.0, 1.0); }
```

```
    public double perimetro() { return 2.0 * PI * r; }
    public double area() { return PI * r * r; }

    // método de objeto para comparar círculos
    public Circulo elMayor(Circulo c) {
        if (this.r>=c.r) return this; else return c;
    }

    // método de clase para comparar círculos
    public static Circulo elMayor(Circulo c, Circulo d) {
        if (c.r>=d.r) return c; else return d;
    }

} // fin de la clase Circulo
```

En este ejemplo se ve cómo se definen las variables miembro y los métodos (cuyos nombres se han resaltado en negrita) dentro de la clase. Dichas variables y métodos pueden ser ***de objeto*** o ***de clase*** (***static***). Se puede ver también cómo el nombre del fichero coincide con el de la clase ***public*** con la extensión ****.java***.

3.3 VARIABLES MIEMBRO

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está ***centrada en los datos***. Una clase está constituida por unos ***datos*** y unos ***métodos*** que operan sobre esos datos.

3.3.1 Variables miembro de objeto

Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro. Las variables miembro de una clase (también llamadas ***campos***) pueden ser de ***tipos primitivos*** (*boolean*, *int*, *long*, *double*, …) o referencias a ***objetos*** de otra clase (*composición*).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de ***tipos primitivos*** se inicializan siempre de modo automático, incluso antes de llamar al ***constructor*** (***false*** para ***boolean***, el carácter nulo para ***char*** y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas también en el constructor.

Las variables miembro pueden también inicializarse explícitamente en la ***declaración***, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

```
long nDatos = 100;
```

Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada ***objeto*** que se crea de una clase tiene ***su propia copia*** de las variables miembro. Por ejemplo, cada objeto de la clase ***Circulo*** tiene sus propias coordenadas del centro ***x*** e ***y***, y su propio valor del radio ***r***.

Los ***métodos de objeto*** se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama ***argumento implícito***. Por ejemplo, para calcular el área de un objeto de la clase ***Circulo*** llamado ***c1*** se escribirá: ***c1.area();***.

Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra ***this*** y el operador punto.

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: ***public***, ***private***, ***protected*** y ***package*** (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (***public*** y ***package***), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro. En el Apartado 3.11, en la página 60, se especifican con detalle las consecuencias de estos modificadores de acceso.

Existen otros dos modificadores (no de acceso) para las variables miembro:

1. ***transient***: indica que esta variable miembro no forma parte de la ***persistencia*** (capacidad de los objetos de mantener su valor cuando termina la ejecución de un programa) de un objeto y por tanto no debe ser ***serializada*** (convertida en flujo de caracteres para poder ser almacenada en disco o en una base de datos) con el resto del objeto.
2. ***volatile***: indica que esta variable puede ser utilizada por distintas ***threads*** sincronizadas (ver Apartado 6.3, en la página 131) y que el compilador no debe realizar optimizaciones con esta variable.

Al nivel de estos apuntes, los modificadores ***transient*** y ***volatile*** no serán utilizados.

3.3.2 Variables miembro de clase (static)

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama ***variables de clase*** o variables ***static***. Las variables ***static*** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo ***PI*** en la clase ***Circulo***) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como ***numCirculos*** en la clase ***Circulo***).

Las variables de clase son lo más parecido que ***Java*** tiene a las ***variables globales*** de C/C++.

Las variables de clase se crean anteponiendo la palabra ***static*** a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, ***Circulo.numCirculos*** es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro ***static*** se inicializan con los valores por defecto para los tipos primitivos (***false*** para ***boolean***, el carácter nulo para ***char*** y cero para los tipos numéricos), y con ***null*** si es una referencia.

Las variables miembro ***static*** se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método ***static*** o en cuanto se utiliza una variable ***static*** de dicha clase. Lo importante es que las variables miembro ***static*** se inicializan siempre antes que cualquier objeto de la clase.

3.4 VARIABLES FINALES

Una variable de un tipo primitivo declarada como ***final*** no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una ***constante***, y equivale a la palabra ***const*** de C/C++.

Java permite separar la ***definición*** de la ***inicialización*** de una variable ***final***. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable ***final*** así definida es ***constante*** (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados ***final***.

Declarar como ***final*** un objeto miembro de una clase hace ***constante*** la ***referencia***, pero no el propio objeto, que puede ser modificado a través de otra referencia. En ***Java*** no es posible hacer que un objeto sea constante.

3.5 MÉTODOS (FUNCIONES MIEMBRO)

3.5.1 Métodos de objeto

Los ***métodos*** son funciones definidas dentro de una clase. Salvo los métodos ***static*** o de clase, se aplican siempre a un objeto de la clase por medio del ***operador punto*** (.). Dicho objeto es su ***argumento implícito***. Los métodos pueden además tener otros ***argumentos explícitos*** que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama ***declaración*** o ***header***; el código comprendido entre las ***llaves*** {…} es el ***cuerpo*** o ***body*** del método. Considérese el siguiente método tomado de la clase ***Circulo***:

```
public Circulo elMayor(Circulo c) {      // header y comienzo del método
    if (this.r>=c.r)                     // body
        return this;                     // body
    else                                 // body
        return c;                        // body
}                                        // final del método
```

El ***header*** consta del cualificador de acceso (***public***, en este caso), del tipo del valor de retorno (***Circulo*** en este ejemplo, ***void*** si no tiene), del ***nombre de la función*** y de una lista de ***argumentos explícitos*** entre paréntesis, separados por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen ***visibilidad directa*** de las variables miembro del objeto que es su ***argumento implícito***, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia ***this***, de modo discrecional (como en el ejemplo anterior con ***this.r***) o si alguna variable local o argumento las oculta.

El ***valor de retorno*** puede ser un valor de un ***tipo primitivo*** o una ***referencia***. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de ***interface***. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.

Los métodos pueden definir ***variables locales***. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

Si en el ***header*** del método se incluye la palabra ***native*** (Ej: `public native void miMetodo();`) no hay que incluir el código o implementación del método. Este código deberá estar en una librería dinámica (*Dynamic Link Library* o DLL). Estas librerías son ficheros de funciones compiladas normalmente en lenguajes distintos de ***Java*** (C, C++, Fortran, etc.). Es la forma de poder utilizar conjuntamente funciones realizadas en otros lenguajes desde código escrito en ***Java***. Este tema queda fuera del caracter fundamentalmente introductorio de este manual.

Un método también puede declararse como ***synchronized*** (Ej: `public synchronized double miMetodoSynch(){...}`). Estos métodos tienen la particularidad de que sobre un objeto no pueden ejecutarse simultáneamente dos métodos que estén sincronizados (véase Apartado 6.3, en la página 131).

3.5.2 Métodos sobrecargados (overloaded)

Al igual que C++, ***Java*** permite métodos ***sobrecargados*** (***overloaded***), es decir, métodos distintos que tienen ***el mismo nombre***, pero que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase ***Circulo*** del Apartado 3.2 presenta dos casos de métodos sobrecargados: los cuatro constructores y los dos métodos llamados ***elMayor()***.

A la hora de llamar a un método sobrecargado, ***Java*** sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más restringido (por ejemplo, *int* en vez de *long*), el programador debe hacer un ***cast*** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la ***sobrecarga*** de métodos es la ***redefinición***. Una clase puede ***redefinir*** (***override***) un método heredado de una superclase. ***Redefinir*** un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la ***herencia***.

3.5.3 Paso de argumentos a métodos

En ***Java*** los argumentos de los ***tipos primitivos*** se pasan siempre ***por valor***. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. Las ***referencias*** se pasan también ***por valor***, pero a través de ellas se pueden modificar los objetos referenciados.

En ***Java*** no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar punteros a función como argumentos). Lo que se puede hacer en ***Java*** es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear ***variables locales*** de los tipos primitivos o referencias. Estas variables locales dejan de existir al terminar la ejecución del método[1]. Los argumentos formales de un método (las variables que aparecen en el ***header*** del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

Si un método devuelve ***this*** (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo,

```
String numeroComoString = ”8.978”;
float  p = Float.valueOf(numeroComoString).floatValue();
```

donde el método ***valueOf(String)*** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float*** sobre el que se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float***. El ejemplo anterior se podía desdoblar en las siguientes sentencias:

```
String numeroComoString = ”8.978”;
Float  f = Float.valueOf(numeroComoString);
float  p = f.floatValue();
```

Obsérvese que se pueden encadenar varias llamadas a métodos por medio del operador punto (.) que, como todos los operadores de ***Java*** excepto los de asignación, se ejecuta de izquierda a derecha (ver Apartado 2.2.11, en la página 30).

3.5.4 Métodos de clase (static)

Análogamente, puede también haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama ***métodos de clase*** o ***static***. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***. Un ejemplo típico de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (*sin(), cos(), exp(), pow(),* etc.). De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

---

[1] En ***Java*** no hay variables locales ***static***, que en C/C++ y Visual Basic son variables locales que conservan su valor entre las distintas llamadas a un método.

Los métodos y variables de clase se crean anteponiendo la palabra ***static***. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, ***Math.sin(ang)***, para calcular el seno de un ángulo).

Los métodos y las variables de clase son lo más parecido que ***Java*** tiene a las funciones y variables globales de C/C++ o Visual Basic.

3.5.5 Constructores

Un punto clave de la *Programación Orientada Objetos* es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. ***Java*** no permite que haya variables miembro que no estén inicializadas[2]. Ya se ha dicho que ***Java*** inicializa siempre con ***valores por defecto*** las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de ***constructores***.

Un ***constructor*** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del ***constructor*** es reservar memoria e inicializar las variables miembro de la clase.

Los ***constructores*** no tienen valor de retorno (ni siquiera ***void***) y su ***nombre*** es el mismo que el de la clase. Su ***argumento implícito*** es el objeto que se está creando.

De ordinario una clase tiene ***varios constructores***, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos ***sobrecargados***). Se llama ***constructor por defecto*** al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un ***constructor*** de una clase puede llamar a ***otro constructor previamente definido*** en la misma clase por medio de la palabra ***this***. En este contexto, la palabra ***this*** sólo puede aparecer en la ***primera sentencia*** de un ***constructor***.

El ***constructor*** de una ***sub-clase*** puede llamar al constructor de su ***super-clase*** por medio de la palabra ***super***, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.

El ***constructor*** es tan importante que, si el programador no prepara ***ningún constructor*** para una clase, el ***compilador*** crea un ***constructor por defecto***, inicializando las variables de los tipos primitivos a su valor por defecto, y los ***Strings*** y las demás ***referencias*** a objetos a ***null***. Si hace falta, se llama al ***constructor*** de la ***super-clase*** para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los ***constructores*** pueden tener también los modificadores de acceso ***public***, ***private***, ***protected*** y ***package***. Si un ***constructor*** es ***private***, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos ***public*** y ***static*** (*factory methods*) que llamen al ***constructor*** y devuelvan un objeto de esa clase.

Dentro de una clase, los ***constructores*** sólo pueden ser llamados por otros ***constructores*** o por métodos ***static***. No pueden ser llamados por los ***métodos de objeto*** de la clase.

---

[2] Sí puede haber variables locales de métodos sin inicializar, pero el compilador da un error si se intentan utilizar sin asignarles previamente un valor.

3.5.6 Inicializadores

Por motivos que se verán más adelante, ***Java*** todavía dispone de una tercera línea de actuación para evitar que haya variables sin inicializar correctamente. Son los ***inicializadores***, que pueden ser ***static*** (para la clase) o ***de objeto***.

3.5.6.1 Inicializadores static

Un ***inicializador static*** es un algo parecido a un método (un bloque {…} de código, sin nombre y sin argumentos, precedido por la palabra ***static***) que se llama automáticamente al crear la clase (al utilizarla por primera vez). También se diferencia del ***constructor*** en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los tipos primitivos pueden inicializarse directamente con asignaciones en la clase o en el constructor, pero para inicializar objetos o elementos más complicados es bueno utilizar un ***inicializador*** (un bloque de código {…}), ya que permite gestionar ***excepciones***[3] con ***try***…***catch***.

Los ***inicializadores static*** se crean dentro de la clase, como métodos sin nombre, sin argumentos y sin valor de retorno, con tan sólo la palabra ***static*** y el código entre llaves {...}. En una clase pueden definirse ***varios inicializadores static***, que se llamarán en el orden en que han sido definidos.

Los ***inicializadores static*** se pueden utilizar para dar valor a las variables ***static***. Además se suelen utilizar para llamar a ***métodos nativos***, esto es, a métodos escritos por ejemplo en C/C++ (llamando a los métodos ***System.load()*** o ***System.loadLibrary()***, que leen las librerías nativas). Por ejemplo:

```
static{
    System.loadLibrary("MyNativeLibrary");
}
```

3.5.6.2 Inicializadores de objeto

A partir de ***Java 1.1*** existen también ***inicializadores de objeto***, que no llevan la palabra ***static***. Se utilizan para las ***clases anónimas***, que por no tener nombre no pueden tener constructor. En este caso, los inicializadores de objeto se llaman cada vez que se crea un objeto de la clase anónima.

3.5.7 Resumen del proceso de creación de un objeto

El proceso de creación de objetos de una clase es el siguiente:

1. Al crear el primer objeto de la clase o al utilizar el primer método o variable ***static*** se localiza la clase y se carga en memoria.
2. Se ejecutan los ***inicializadores static*** (sólo una vez).
3. Cada vez que se quiere crear un ***nuevo objeto***:
   - se comienza reservando la memoria necesaria

[3] Las ***excepciones*** son situaciones de error o, en general, situaciones anómalas que puede exigir ciertas actuaciones del propio programa o del usuario. Las excepciones se explican con más detalle en el Capítulo 8.

- se da valor por defecto a las variables miembro de los tipos primitivos
- se ejecutan los inicializadores de objeto
- se ejecutan los constructores

3.5.8 Destrucción de objetos (liberación de memoria)

En ***Java*** no hay ***destructores*** como en C++. El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han ***perdido la referencia***, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo por haber llegado al final del bloque en el que habían sido definidos, porque a la ***referencia*** se le ha asignado el valor ***null*** o porque a la ***referencia*** se le ha asignado la dirección de otro objeto. A esta característica de ***Java*** se le llama ***garbage collection*** (recogida de basura).

En ***Java*** es normal que varias variables de tipo referencia apunten al mismo objeto. ***Java*** lleva internamente un contador de cuántas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero. Como ya se ha dicho, una forma de hacer que un objeto quede sin referencia es cambiar ésta a ***null***, haciendo por ejemplo:

```
ObjetoRef = null;
```

En ***Java*** no se sabe exactamente cuándo se va a activar el ***garbage collector***. Si no falta memoria es posible que no se llegue a activar en ningún momento. No es pues conveniente confiar en él para la realización de otras tareas más críticas.

Se puede llamar explícitamente al ***garbage collector*** con el método ***System.gc()***, aunque esto es considerado por el sistema sólo como una "sugerencia" a la JVM.

3.5.9 Finalizadores

Los ***finalizadores*** son métodos que vienen a completar la labor del ***garbage collector***. Un ***finalizador*** es un método que se llama automáticamente cuando se va a destruir un objeto (antes de que la memoria sea liberada de modo automático por el sistema). Se utilizan para ciertas ***operaciones de terminación*** distintas de liberar memoria (por ejemplo: cerrar ficheros, cerrar conexiones de red, liberar memoria reservada por funciones nativas, etc.). Hay que tener en cuenta que el ***garbage collector*** sólo libera la memoria reservada con ***new***. Si por ejemplo se ha reservado memoria con funciones nativas en C (por ejemplo, utilizando la función ***malloc()***), esta memoria hay que liberarla explícitamente utilizando el método ***finalize()***.

Un ***finalizador*** es un método de objeto (no ***static***), sin valor de retorno (***void***), sin argumentos y que siempre se llama ***finalize()***. Los ***finalizadores*** se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un ***finalizador*** debería terminar siempre llamando al ***finalizador*** de su ***super-clase***.

Tampoco se puede saber el momento preciso en que los ***finalizadores*** van a ser llamados. En muchas ocasiones será conveniente que el programador realice esas operaciones de finalización de modo explícito mediante otros métodos que él mismo llame.

El método ***System.runFinalization()*** "sugiere" a la JVM que ejecute los ***finalizadores*** de los objetos pendientes (que han perdido la referencia). Parece ser que para que este método se ejecute, en Java 1.1 hay que llamar primero a ***gc()*** y luego a ***runFinalization()***.

3.6 PACKAGES

3.6.1 Qué es un package

Un ***package*** es una agrupación de clases. En la API de ***Java 1.1*** había 22 ***packages***; en ***Java 1.2*** hay 59 ***packages***, lo que da una idea del "crecimiento" experimentado por el lenguaje.

Además, el usuario puede crear sus propios ***packages***. Para que una clase pase a formar parte de un ***package*** llamado ***pkgName***, hay que introducir en ella la sentencia:

```
package pkgName;
```

que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.

Los nombres de los ***packages*** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un ***package*** puede constar de varios nombres unidos por puntos (los propios ***packages*** de ***Java*** siguen esta norma, como por ejemplo ***java.awt.event***).

Todas las clases que forman parte de un ***package*** deben estar en el mismo directorio. Los nombres compuestos de los ***packages*** están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los ***nombres de las clases*** de ***Java*** sean únicos en ***Internet***. Es el nombre del ***package*** lo que permite obtener esta característica. Una forma de conseguirlo es incluir el ***nombre del dominio*** (quitando quizás el país), como por ejemplo en el ***package*** siguiente:

```
es.ceit.jgjalon.infor2.ordenar
```

Las clases de un ***package*** se almacenan en un directorio con el mismo nombre largo (***path***) que el ***package***. Por ejemplo, la clase,

```
es.ceit.jgjalon.infor2.ordenar.QuickSort.class
```

debería estar en el directorio,

```
CLASSPATH\es\ceit\jgjalon\infor2\ordenar\QuickSort.class
```

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de ***Java*** (clases del sistema o de usuario), en este caso la posición del directorio ***es*** en los discos locales del ordenador.

Los ***packages*** se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.
2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de ***Java*** es la ***Internet***). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del ***package***.
3. Para ayudar en el control de la accesibilidad de clases y miembros.

3.6.2 Cómo funcionan los packages

Con la sentencia ***import packname;*** se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre

nombres de clases, ***Java*** da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del ***package***.

El importar un ***package*** no hace que se carguen todas las clases del ***package***: sólo se cargarán las clases ***public*** que se vayan a utilizar. Al importar un ***package*** no se importan los ***sub-packages***. Éstos deben ser importados explícitamente, pues en realidad son ***packages*** distintos. Por ejemplo, al importar ***java.awt*** no se importa ***java.awt.event***.

Es posible guardar en jerarquías de directorios diferentes los ficheros ****.class*** y ****.java***, con objeto por ejemplo de no mostrar la situación del código fuente. Los ***packages*** hacen referencia a los ficheros compilados ****.class***.

En un programa de ***Java***, una clase puede ser referida con su nombre completo (el nombre del ***package*** más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia ***import*** permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del ***package*** importado. Se importan por defecto el ***package java.lang*** y el ***package*** actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar ***import***: para ***una clase*** y para ***todo un package***:

```
import es.ceit.jgjalon.infor2.ordenar.QuickSort.class;
import es.ceit.jgjalon.infor2.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\es\ceit\jgjalon\infor2\ordenar
```

El cómo afectan los ***packages*** a los permisos de acceso de una clase se estudia en el Apartado 3.11, en la página 60.

3.7 HERENCIA

3.7.1 Concepto de herencia

Se puede construir una clase a partir de otra mediante el mecanismo de la ***herencia***. Para indicar que una clase deriva de otra se utiliza la palabra ***extends***, como por ejemplo:

```
class CirculoGrafico extends Circulo {...}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser ***redefinidas*** (***overridden***) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la ***sub-clase*** (la clase derivada) "contuviera" un objeto de la ***super-clase***; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de ***Java*** creadas por el programador tienen una ***super-clase***. Cuando no se indica explícitamente una ***super-clase*** con la palabra ***extends***, la clase deriva de ***java.lang.Object***,

que es la clase raíz de toda la jerarquía de clases de ***Java***. Como consecuencia, todas las clases tienen algunos métodos que han heredado de ***Object***.

La ***composición*** (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la ***herencia*** en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace ***private***).

3.7.2 La clase Object

Como ya se ha dicho, la clase ***Object*** es la raíz de toda la jerarquía de clases de ***Java***. Todas las clases de ***Java*** derivan de ***Object***.

La clase ***Object*** tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos se pueden citar los siguientes:

1. Métodos que pueden ser redefinidos por el programador:

   clone() Crea un objeto a partir de otro objeto de la misma clase. El método original heredado de ***Object*** lanza una *CloneNotSupportedException*. Si se desea poder clonar una clase hay que implementar la interface ***Cloneable*** y redefinir el método ***clone()***. Este método debe hacer una copia miembro a miembro del objeto original. No debería llamar al operador ***new*** ni a los constructores.

   equals() Indica si dos objetos son o no iguales. Devuelve ***true*** si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro.

   toString() Devuelve un ***String*** que contiene una representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.

   finalize() Este método ya se ha visto al hablar de los ***finalizadores***.

2. Métodos que no pueden ser redefinidos (son métodos ***final***):

   getClass() Devuelve un objeto de la clase ***Class***, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su super-clase, las interfaces implementadas, etc. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.

   notify(), ***notifyAll()*** y ***wait()*** Son métodos relacionados con las ***threads*** y se verán en el Capítulo 6.

3.7.3 Redefinición de métodos heredados

Una clase puede ***redefinir*** (volver a definir) cualquiera de los métodos heredados de su ***super-clase*** que no sean ***final***. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Las métodos de la ***super-clase*** que han sido redefinidos pueden ser todavía accedidos por medio de la palabra ***super*** desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden ***ampliar los derechos de acceso*** de la ***super-clase*** (por ejemplo ser ***public***, en vez de ***protected*** o ***package***), pero nunca restringirlos.

Los ***métodos de clase*** o ***static*** no pueden ser redefinidos en las clases derivadas.

3.7.4 Clases y métodos abstractos

Una ***clase abstracta*** (***abstract***) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra ***abstract***, como por ejemplo,

```
public abstract class Geometria { ... }
```

Una clase ***abstract*** puede tener métodos declarados como ***abstract***, en cuyo caso no se da definición del método. Si una clase tiene algún método ***abstract*** es obligatorio que la clase sea ***abstract***. En cualquier ***sub-clase*** este método deberá bien ser redefinido, bien volver a declararse como ***abstract*** (el método y la ***sub-clase***).

Una clase ***abstract*** puede tener métodos que no son ***abstract***. Aunque no se puedan crear objetos de esta clase, sus ***sub-clases*** heredarán el método completamente a punto para ser utilizado.

Como los métodos ***static*** no pueden ser redefinidos, un método ***abstract*** no puede ser ***static***.

3.7.5 Constructores en clases derivadas

Ya se comentó que un ***constructor*** de una clase puede llamar por medio de la palabra ***this*** a otro ***constructor*** previamente definido en la misma clase. En este contexto, la palabra ***this*** sólo puede aparecer en la primera sentencia de un ***constructor***.

De forma análoga el ***constructor*** de una clase derivada puede llamar al ***constructor*** de su ***super-clase*** por medio de la palabra ***super()***, seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la ***super-clase***. De esta forma, un ***constructor*** sólo tiene que inicializar directamente las variables no heredadas.

La llamada al ***constructor*** de la ***super-clase*** debe ser la ***primera sentencia del constructor***[4], excepto si se llama a otro constructor de la misma clase con ***this()***. Si el programador no la incluye, ***Java*** incluye automáticamente una llamada al ***constructor por defecto*** de la ***super-clase***, ***super()***. Esta llamada en cadena a los ***constructores de las super-clases*** llega hasta el origen de la jerarquía de clases, esto es al constructor de ***Object***.

Como ya se ha dicho, si el programador no prepara un ***constructor por defecto***, el compilador crea uno, inicializando las variables de los ***tipos primitivos*** a sus valores por defecto, y los ***Strings*** y demás ***referencias*** a objetos a ***null***. Antes, incluirá una llamada al constructor de la ***super-clase***.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con ***new***, de la que se encarga el ***garbage collector***), es importante llamar a los ***finalizadores*** de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el ***finalizador de la sub-clase*** deba realizar todas sus tareas primero y luego llamar al finalizador de la

---

[4] De todas formas, antes de ejecutar esa llamada ya se ha reservado la memoria necesaria para crear el objeto y se han inicializado las variables miembro (ver Apartado 3.5.7).

super-clase en la forma ***super.finalize()***. Los métodos ***finalize()*** deben ser al menos ***protected***, ya que el método ***finalize()*** de ***Object*** lo es, y no está permitido reducir los permisos de acceso en la herencia.

3.8 CLASES Y MÉTODOS FINALES

Recuérdese que las ***variables*** declaradas como ***final*** no pueden cambiar su valor una vez que han sido inicializadas. En este apartado se van a presentar otros dos usos de la palabra ***final***.

Una ***clase*** declarada ***final*** no puede tener clases derivadas. Esto se puede hacer por motivos de ***seguridad*** y también por motivos de ***eficiencia***, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

Análogamente, un ***método*** declarado como ***final*** no puede ser redefinido por una clase que derive de su propia clase.

3.9 INTERFACES

3.9.1 Concepto de interface

Una ***interface*** es un conjunto de ***declaraciones de métodos*** (sin ***definición***). También puede definir ***constantes***, que son implícitamente ***public***, ***static*** y ***final***, y deben siempre inicializarse en la declaración. Estos métodos definen un *tipo de conducta*. Todas las clases que implementan una determinada ***interface*** están obligadas a proporcionar una definición de los métodos de la ***interface***, y en ese sentido adquieren una ***conducta*** o ***modo de funcionamiento***.

Una ***clase*** puede ***implementar*** una o varias ***interfaces***. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las ***interfaces***, separados por comas, detrás de la palabra ***implements***, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico extends Circulo
                            implements Dibujable, Cloneable {
    ...
}
```

¿Qué diferencia hay entre una ***interface*** y una ***clase abstract***? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase ***abstract*** puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas ***diferencias importantes***:

1. Una clase no puede heredar de dos clases ***abstract***, pero sí puede heredar de una clase ***abstract*** e implementar una ***interface***, o bien implementar dos o más ***interfaces***.
2. Una clase no puede heredar métodos -definidos- de una ***interface***, aunque sí ***constantes***.
3. Las ***interfaces*** permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, inpendientemente de su situación en la jerarquía de clases de ***Java***.

4. Las ***interfaces*** permiten "publicar" el comportamiento de una clase desvelando un mínimo de información.
5. Las ***interfaces*** tienen una ***jerarquía*** propia, independiente y más flexible que la de las clases, ya que tienen permitida la ***herencia múltiple***.
6. De cara al ***polimorfismo*** (recordar el Apartado 1.3.8, a partir de la página 15), las ***referencias*** de un tipo ***interface*** se pueden utilizar de modo similar a las clases ***abstract***.

3.9.2 Definición de interfaces

Una ***interface*** se define de un modo muy similar a las clases. A modo de ejemplo se reproduce aquí la definición de la interface ***Dibujable*** dada en el Apartado 1.3.5:, en la página 13:

```
// fichero Dibujable.java

import java.awt.Graphics;

public interface Dibujable {
    public void setPosicion(double x, double y);
    public void dibujar(Graphics dw);
}
```

Cada ***interface public*** debe ser definida en un fichero ****.java*** con el mismo nombre de la ***interface***. Los ***nombres de las interfaces*** suelen comenzar también con ***mayúscula***.

Las ***interfaces*** no admiten más que los modificadores de acceso ***public*** y ***package***. Si la ***interface*** no es ***public*** no será accesible desde fuera del ***package*** (tendrá la accesibilidad por defecto, que es ***package***). Los métodos declarados en una ***interface*** son siempre ***public*** y ***abstract***, de modo implícito.

3.9.3 Herencia en interfaces

Entre las ***interfaces*** existe una ***jerarquía*** (independiente de la de las clases) que permite ***herencia simple y múltiple***. Cuando una ***interface*** deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una ***interface*** puede derivar de varias ***interfaces***. Para la herencia de ***interfaces*** se utiliza asimismo la palabra ***extends***, seguida por el nombre de las ***interfaces*** de las que deriva, separadas por comas.

Una ***interface*** puede ocultar una constante definida en una ***super-interface*** definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una ***super-interface***.

Las ***interfaces*** no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha ***interface*** dejarán de funcionar, a menos que implementen el nuevo método.

3.9.4 Utilización de interfaces

Las ***constantes*** definidas en una ***interface*** se pueden utilizar en cualquier clase (aunque no implemente la ***interface***) precediéndolas del nombre de la ***interface***, como por ejemplo (suponiendo que PI hubiera sido definida en ***Dibujable***):

```
area = 2.0*Dibujable.PI*r;
```

Sin embargo, en las clases que ***implementan*** la ***interface*** las constantes se pueden utilizar directamente, como si fueran constantes de la clase. A veces se crean interfaces para agrupar constantes simbólicas relacionadas (en este sentido pueden en parte suplir las variables ***enum*** de C/C++).

De cara al ***polimorfismo***, el nombre de una ***interface*** se puede utilizar como un ***nuevo tipo de referencia***. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la ***interface***. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

3.10 CLASES INTERNAS

Una ***clase interna*** es una clase definida dentro de otra clase, llamada ***clase contenedora***, en alguna variante de la siguiente forma general:

```
class ClaseContenedora {
    ...
    class ClaseInterna {
        ...
    }
    ...
}
```

Las ***clases internas*** fueron introducidas en la versión ***Java 1.1***. Además de su utilidad en sí, las ***clases internas*** se utilizan mucho en el nuevo ***modelo de eventos*** que se introdujo en dicha versión de ***Java***.

Hay cuatro tipos de ***clases internas***:

1. Clases internas *static*.
2. Clases internas *miembro*.
3. Clases internas *locales*.
4. Clases *anónimas*.

En lo sucesivo se utilizará la terminología ***clase contenedora*** o ***clase global*** para hacer referencia a la clase que contiene a la ***clase interna***.

Hay que señalar que la JVM (***Java Virtual Machine***) no sabe nada de la existencia de ***clases internas***. Por ello, el compilador convierte estas clases en ***clases globales***, contenidas en ficheros ****.class*** cuyo nombre es ***ClaseContenedora$ClaseInterna.class***. Esta conversión inserta variables ocultas, métodos y argumentos en los constructores. De todas formas, lo que más afecta al programador de todo esto es lo referente al nombre de los ficheros que aparecen en el directorio donde se realiza la compilación, que pueden resultar sorprendentes si no se conoce su origen.

3.10.1 Clases e interfaces internas static

Se conocen también con el nombre de ***clases anidadas*** (*nested classes*). Las clases e interfaces internas static sólo pueden ser creadas dentro de otra clase ***al máximo nivel***, es decir directamente

en el bloque de definición de la ***clase contenedora*** y no en un bloque más interno. Es posible definir ***clases e interfaces internas static*** dentro de una ***interface contenedora***. Este tipo de clases internas se definen utilizando la palabra ***static***. Todas las ***interfaces internas*** son implícitamente ***static***.

En cierta forma, las ***clases internas static*** se comportan como clases normales en un ***package***. Para utilizar su nombre desde fuera de la clase contenedora hay que precederlo por el ***nombre de la clase contenedora*** y el ***operador punto*** (.) Este tipo de relación entre clases se puede utilizar para agrupar varias clases dentro de una clase más general. Lo mismo puede decirse de las ***interfaces internas***.

Las ***clases internas static*** pueden ver y utilizar los miembros ***static*** de la ***clase contenedora***. No se necesitan objetos de la clase contenedora para crear objetos de la ***clase interna static***. Los métodos de la ***clase interna static*** no pueden acceder directamente a los objetos de la clase contenedora, caso de que los haya: deben disponer de una referencia a dichos objetos, como cualquier otra clase.

La sentencia ***import*** puede utilizarse para importar una ***clase interna static***, en la misma forma que si se tratara de importar una clase de un ***package*** (con el punto (.)). Por ejemplo, si la interface ***Linkable*** es interna a la clase ***List***, para implementar dicha interface hay que escribir,

```
... implements List.Linkable
```

y para importarla hay que usar,

```
import List.*; // o bien
import List.Linkable;
```

Otras características importantes son las siguientes:

1. Pueden definirse ***clases*** e ***interfaces internas*** dentro de ***interface*** y ***clases contenedoras***, con las cuatro combinaciones posibles.
2. Puede haber ***varios niveles***, esto es una ***clase interna static*** puede ser ***clase contenedora*** de otra ***clase interna static***, y así sucesivamente.
3. Las ***clases e interfaces internas static*** pertenecen al ***package*** de la ***clase contenedora***.
4. Pueden utilizarse los calificadores ***final***, ***public***, ***private*** y ***protected***. Ésta es una forma más de controlar el acceso a ciertas clases.

A continuación se presenta un ejemplo de ***clase interna static***:

```
// fichero ClasesIntStatic.java

class A {
    int i=1;            // variable miembro de objeto
    static int is=-1;   // variable miembro de clase
    public A(int i) {this.i=i;} // constructor
    // a los métodos de la clase contenedora hay que pasarles referencias
    //     a los objetos de la clase interna static
    public void printA(Bs unBs) {
        System.out.println("i="+i+" unBs.j="+unBs.j);
    }
    // definición de una clase interna static
    static class Bs {
        int j=2;
        public Bs(int j) {this.j=j;} // constructor
```

```
        // los métodos de la clase interna static no pueden acceder a la i
        //     pues es una variable de objeto. Sí pueden acceder a is
        public void printBs() {
            System.out.println(" j=" + j + " is=" + is);
        }
    } // fin clase Bs

} // fin clase contenedora A


class ClasesIntStatic {
    public static void main(String [] arg) {
        A a1 = new A(11), a2 = new A(12);
        println("a1.i=" + a1.i + "  a2.i=" + a2.i);
        // dos formas de crear objetos de la clase interna static
        A.Bs b1 = new A.Bs(-10);  // necesario poner A.Bs
        A.Bs b2 = a1.new Bs(-11); // b2 es independiente de a1
        // referencia directa a los objetos b1 y b2
        println("b1.j=" + b1.j + " b2.j=" + b2.j);

        // los métodos de la clase interna acceden directamente a las variables
        //     de la clase contenedora sólo si son static
        b1.printBs();  // escribe:   j=-10 is=-1
        b2.printBs();  // escribe:   j=-20 is=-1

        // a los métodos de la clase contenedora hay que pasarles referencias
        //     a los objetos de la clase interna, pera que puedan identificarlos
        a1.printA(b1); // escribe: i=11 unBs.j=-10
        a1.printA(b2); // escribe: i=11 unBs.j=-11

    } // fin de main()

    public static void println(String str) {System.out.println(str);}

}  // fin clase ClasesIntStatic
```

3.10.2 Clases internas miembro (no static)

Las ***clases internas miembro*** o simplemente ***clases internas***, son clases definidas al máximo nivel de la ***clase contenedora*** (directamente en el bloque de diminución de dicha clase), sin la palabra ***static***. Se suelen llamar ***clases internas miembro*** o simplemente ***clases internas***. No existen ***interfaces internas*** de este tipo.

Las ***clases internas*** no pueden tener ***variables miembro static***. Tienen una nueva sintaxis para las palabras ***this***, ***new*** y ***super***, que se verá un poco más adelante.

La característica principal de estas clases internas e que cada objeto de la ***clase interna*** existe siempre dentro de un y sólo un objeto de la ***clase contenedora***. Un objeto de la ***clase contenedora*** puede estar relacionado con uno o más objetos de la ***clase interna***. Tener esto presente es muy importante para entender las características que se explican a continuación.

Relación entre las ***clases interna*** y ***contenedora*** respecto al ***acceso a las variables miembro***:

1. Debido a la relación uno a uno, los métodos de la clase interna ven directamente las variables miembro del objeto de la clase contenedora, sin necesidad de cualificarlos.
2. Sin embargo, los métodos de la clase contenedora no ven directamente las variables miembro de los objetos de la clase interna: necesitan cualificarlos con una referencia a los

correspondientes objetos. Esto es consecuencia de la relación uno a varios que existe entre los objetos de la clase contenedora y los de la clase interna.

3. ***Otras clases diferentes*** de las ***clases contenedora*** e ***interna*** pueden utilizar directamente los ***objetos de la clase interna***, sin cualificarlos con el ***objeto o el nombre de la clase contenedora***. De hecho, se puede seguir accediendo a los objetos de la clase interna aunque se pierda la referencia al objeto de la clase contenedora con el que están asociados.

Respecto a los ***permisos de acceso***:

1. Las ***clases internas*** pueden también ser ***private*** y ***protected*** (las clases normales sólo pueden ser ***public*** y ***package***). Esto permite nuevas posibilidades de encapsulación.
2. Los métodos de las ***clases internas*** acceden directamente a todos los miembros, incluso ***private***, de la ***clase contenedora***.
3. También la ***clase contenedora*** puede acceder –si dispone de una referencia- a todas las variables miembro (incluso ***private***) de sus ***clases internas***.
4. Una ***clase interna*** puede acceder también a los miembros (incluso ***private***) de otras ***clases internas*** definidas en la misma ***clase contenedora***.

Otras características de las ***clases internas*** son las siguientes:

1. Una ***clase interna miembro*** puede contener otra ***clase interna miembro***, hasta el nivel que se desee (aunque no se considera buena técnica de programación utilizar muchos niveles).
2. En la ***clase interna***, la palabra ***this*** se refiere al objeto de la propia ***clase interna***. Para acceder al objeto de la ***clase contenedora*** se utiliza ***ClaseContenedora.this***.
3. Para crear un nuevo objeto de la ***clase interna*** se puede utilizar ***new***, precedido por la referencia al objeto de la ***clase contenedora*** que contendrá el nuevo objeto: ***unObjCC.new()***. El ***tipo del objeto*** es el nombre de la clase contenedora seguido del nombre de la clase interna, como por ejemplo:

```
ClaseCont.ClaseInt unObjClInt = unObjClaCont.new ClaseInt(...);
```

4. Supóngase como ejemplo adicional que B es una clase interna de A y que C es una clase interna de B. La creación de objetos de las tres clases se puede hacer del siguiente modo:

```
A a = new A();          // se crea un objeto de la clase A
A.B b = a.new B();      // b es un objeto de la clase interna B dentro de a
A.B.C c = b.new C();    // c es un objeto de la clase interna C dentro de b
```

5. Nunca se puede crear un objeto de la ***clase interna*** sin una referencia a un objeto de la ***clase contenedora***. Los ***constructores*** de la ***clase interna*** tienen como argumento oculto una referencia al objeto de la ***clase contenedora***.
6. El nuevo significado de la palabra ***super*** es un poco complicado: Si una clase deriva de una ***clase interna***, su constructor no puede llamar a ***super()*** directamente. Ello hace que el compilador no pueda crear un ***constructor por defecto***. Al constructor hay que pasarle una referencia a la ***clase contenedora*** de la ***clase interna super-clase***, y con esa referencia ***ref*** llamar a ***ref.super()***.

Las ***clases internas*** pueden ***derivar*** de otras clases diferentes de la ***clase contenedora***. En este caso, conviene tener en cuenta las siguientes reglas:

1. Las ***clases internas*** constituyen como una ***segunda jerarquía de clases*** en ***Java***: por una parte están en la ***clases contenedora*** y ven sus variables; por otra parte ***pueden derivar de otra clase*** que no tenga nada que ver con la ***clase contenedora***. Es muy importante evitar conflictos con los nombres. En caso de conflicto entre un nombre heredado y un nombre en la clase contenedora, el nombre heredado debe tener prioridad.
2. En caso de conflicto de nombres, ***Java*** obliga a utilizar la referencia ***this*** con un nuevo significado: para referirse a la ***variable o método miembro heredado*** se utiliza ***this.name***, mientras que se utiliza ***NombreClaseCont.this.name*** para el ***miembro de la clase contenedora***..
3. Si una ***clase contenedora*** deriva de una ***super-clase*** que tiene una ***clase interna***, la ***clase interna de la sub-clase*** puede a su vez derivar de la ***clase interna de la super-clase*** y redefinir todos los métodos que necesite. La casuística se puede complicar todo lo que se desee, pero siempre hay que recomendar hacer las cosas lo más sencillas que sea posible.

El uso de las ***clases internas miembro*** tiene las siguientes restricciones:

1. Las ***clases internas*** no pueden tener el mismo nombre que la ***clase contenedora*** o ***package***.
2. Tampoco pueden tener miembros ***static***: variables, métodos o clases.

A continuación se presenta un ejemplo completo de utilización de ***clases internas miembro***:

```
// fichero ClasesInternas.java

// clase contenedora
class A {
    int i=1;                  // variable miembro
    public A(int i) {this.i=i;}  // constructor
    // los métoodos de la clase contenedora necesitan una
    //     referencia a los objetos de la clase interna
    public void printA(B unB) {
        System.out.println("i="+i+" unB.j="+unB.j); // sí acepta unB.j
    }
    // la clase interna puede tener cualquier visibilidad. Con private da error
    //     porque main() no puede acceder a la clase interna
    protected class B {
        int j=2;
        public B(int j) {this.j=j;} // constructor
        public void printB() {
            System.out.println("i=" + i + " j=" + j); // sí sabe qué es j
        }
    } // fin clase B
} // fin clase contenedora A


class ClasesInternas {
    public static void main(String [] arg) {
        A a1 = new A(11); A a2 = new A(12);
        println("a1.i=" + a1.i + "  a2.i=" + a2.i);
        // forma de crear objetos de la clase interna
        //     asociados a un objeto de la clase contenedora
        A.B b1 = a1.new B(-10), b2 = a1.new B(-20);
        // referencia directa a los objetos b1 y b2 (sin cualificar).
        println("b1.j=" + b1.j + " b2.j=" + b2.j);
```

```
        // los métodos de la clase interna pueden acceder directamente a
        //    las variables miembro del objeto de la clase contenedora
        b1.printB();  // escribe:   i=11 j=-10
        b2.printB();  // escribe:   i=11 j=-20
        // los métodos de la clase contenedora deben recibir referencias
        //     a los objetos de la clase interna, pera que puedan identificarlos
        a1.printA(b1); a1.printA(b2);

        A a3 = new A(13);
        A.B b3 = a3.new B(-30);
        println("b3.j=" + b3.j);
        a3 = null; // se destruye la referencia al objeto de la clase contenedora
        b3.printB();    // escribe:   i=13 j=-30
        a3 = new A(14); // se crea un nuevo objeto asociado a la referencia a3
        // b3 sigue asociado an anterior objeto de la clase contenedora
        b3.printB();  // escribe:   i=13 j=-30
    } // fin de main()

    public static void println(String str) {System.out.println(str);}
} // fin clase ClasesInternas
```

3.10.3 Clases internas locales

Las ***clases internas locales*** o simplemente ***clases locales*** no se declaran dentro de otra clase al máximo nivel, sino dentro de un ***bloque de código***, normalmente en un ***método***, aunque también se pueden crear en un ***inicializador static*** o de objeto.

Las principales características de las ***clases locales*** so las siguientes:

1. Como las ***variables locales***, las ***clases locales*** sólo son visibles y utilizables en el bloque de código en el que están definidas. Los objetos de la ***clase local*** deben ser creados en el mismo bloque en que dicha clase ha sido definida. De esta forma se puede acercar la definición al uso de la clase.
2. Las ***clases internas locales*** tienen acceso a todas las variables miembro y métodos de la ***clase contenedora***. Pueden ver también los ***miembros heredados***, tanto por la ***clase interna local*** como por la ***clase contenedora***.
3. Las ***clases locales*** pueden utilizar las variables locales y argumentos de métodos ***visibles en ese bloque de código***, pero sólo si son ***final***[5] (en realidad la ***clase local*** trabaja con sus copias de las ***variables locales*** y por eso se exige que sean ***final*** y no puedan cambiar).
4. Un objeto de una ***clase interna local*** sólo puede existir en relación con un objeto de la ***clase contenedora***, que debe existir previamente.
5. La palabra ***this*** se puede utilizar en la misma forma que en las ***clases internas*** miembro, pero no las palabras ***new*** y ***super***.

Restricciones en el uso de las ***clases internas locales***:

1. No pueden tener el mismo nombre que ninguna de sus clases contenedoras.
2. No pueden definir variables, métodos y clases ***static***.

[5] En Java 1.0 el cualificador ***final*** podía aplicarse a variables miembro, métodos y clases. En Java 1.1 puede también aplicarse a variables locales, argumentos de métodos e incluso al argumento de una ***exception***.

3. No pueden ser declaradas ***public***, ***protected***, ***private*** o ***package***, pues su visibilidad es siempre la de las variables locales, es decir, la del bloque en que han sido definidas.

Las ***clases internas locales*** se utilizan para definir clases ***Adapter*** en el AWT. A continuación se presenta un ejemplo de definición de clases internas locales:

```
// fichero ClasesIntLocales.java
// Este fichero demuestra cómo se crean clases locales

class A {
    int i=-1;    // variable miembro
    // constructor
    public A(int i) {this.i=i;}

    // definición de un método de la clase A
    public void getAi(final long k) {    // argumento final
        final double f=3.14;             // variable local final
        // definición de una clase interna local
        class BL {
            int j=2;
            public BL(int j) {this.j=j;} // constructor
            public void printBL() {
                System.out.println(" j="+j+" i="+i+" f="+f+" k="+k);
            }
        } // fin clase BL
        // se crea un objeto de BL
        BL bl = new BL(2*i);
        // se imprimen los datos de ese objeto
        bl.printBL();
    } // fin getAi
} // fin clase contenedora A


class ClasesIntLocales {
    public static void main(String [] arg) {
        // se crea dos objetos de la clase contenedora
        A a1 = new A(-10);
        A a2 = new A(-11);
        // se llama al método getAi()
        a1.getAi(1000); // se crea y accede a un objeto de la clase local
        a2.getAi(2000);
    } // fin de main()

    public static void println(String str) {System.out.println(str);}

}  // fin clase ClasesIntLocales
```

3.10.4 Clases anónimas

Las ***clases anónimas*** son muy similares a las ***clases internas locales***, pero ***sin nombre***. En las ***clases internas locales*** primero se define la clase y luego se crean uno o más objetos. En las ***clases anónimas*** se unen estos dos pasos: Como la clase no tiene nombre sólo se puede crear un único objeto, ya que las ***clases anónimas*** no pueden definir ***constructores***. Las clases anónimas se utilizan con mucha frecuencia en el AWT para definir clases y objetos que gestionen los eventos de los distintos componentes de la interface de usuario. No hay ***interfaces anónimas***.

Formas de definir una ***clase anónima***:

1. Las ***clases anónimas*** requieren una extensión de la palabra clave ***new***. Se definen en una expresión de ***Java***, incluida en una asignación o en la llamada a un método. Se incluye la palabra ***new*** seguida de la ***definición de la clase anónima***, entre llaves {…}.
2. Otra forma de definirlas es mediante la palabra ***new*** seguida del nombre de la clase de la que hereda (sin ***extends***) y la definición de la ***clase anónima*** entre llaves {…}. El nombre de la ***super-clase*** puede ir seguido de argumentos para su ***constructor*** (entre paréntesis, que con mucha frecuencia estarán vacíos pues se utilizará un constructor por defecto).
3. Una tercera forma de definirlas es con la palabra ***new*** seguida del nombre de la ***interface*** que implementa (sin ***implements***) y la definición de la ***clase anónima*** entre llaves {…}. En este caso la ***clase anónima*** deriva de ***Object***. El nombre de la ***interface*** va seguido por paréntesis vacíos, pues el constructor de ***Object*** no tiene argumentos.

Para las ***clases anónimas compiladas*** el compilador produce ficheros con un nombre del tipo ***ClaseContenedora$1.class***, asignando un número correlativo a cada una de las ***clases anónimas***.

Conviene ser muy cuidadoso respecto a los aspectos tipográficos de la definición de ***clases anónimas***, pues al no tener nombre dichas clases suelen resultar difíciles de leer e interpretar. Se aconseja utilizar las siguientes normas ***tipográficas***:

1. Se aconseja que la palabra ***new*** esté en la misma línea que el resto de la expresión.
2. Las ***llaves*** se abren en la misma línea que ***new***, después del cierre del paréntesis de los argumentos del constructor.
3. El ***cuerpo de la clase anónima*** se debe sangrar o indentar respecto a las líneas anteriores de código para que resulte claramente distinguible.
4. El ***cierre de las llaves*** va seguido por el ***resto de la expresión*** en la que se ha definido la ***clase anónima***. Esto puede servir como indicación tipográfica del cierre. Puede ser algo así como **};** o **});**

A continuación se presenta un ejemplo de definición de clase anónima en relación con el AWT:

```
unObjeto.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ...
    }
});
```

donde en negrita se señala la ***clase anónima***, que deriva de ***Object*** e implementa la interface ***ActionListener***.

Las ***clases anónimas*** se utilizan en lugar de ***clases locales*** para clases con muy poco código, de las que sólo hace falta un objeto. No pueden tener ***constructores***, pero sí ***inicializadores static*** o de ***objeto***. Además de las restricciones citadas, tienen ***restricciones*** similares a las ***clases locales***.

3.11 PERMISOS DE ACCESO EN JAVA

Una de las características de la *Programación Orientada a Objetos* es la ***encapsulación***, que consiste básicamente en ocultar la información que no es pertinente o necesaria para realizar una

determinada tarea. Los permisos de acceso de ***Java*** son una de las herramientas para conseguir esta finalidad.

3.11.1 Accesibilidad de los packages

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un ***package*** es accesible si sus directorios y ficheros son accesibles (si están en un ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos ***packages*** que se encuentren en la variable ***CLASSPATH*** del sistema.

3.11.2 Accesibilidad de clases o interfaces

En principio, cualquier ***clase*** o ***interface*** de un ***package*** es accesible para todas las demás clases del ***package***, tanto si es ***public*** como si no lo es. Una clase ***public*** es accesible para cualquier otra clase siempre que su ***package*** sea accesible. Recuérdese que las ***clases*** e ***interfaces*** sólo pueden ser ***public*** o ***package*** (la opción por defecto cuando no se pone ningún modificador).

3.11.3 Accesibilidad de las variables y métodos miembros de una clase:

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia ***this***) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.
2. Los miembros ***private*** de una clase sólo son accesibles para la propia clase.
3. Si el ***constructor*** de una clase es ***private***, sólo un método ***static*** de la propia clase puede crear objetos.

Desde una ***sub-clase***:

1. Las ***sub-clases*** heredan los miembros ***private*** de su ***super-clase***, pero sólo pueden acceder a ellos a través de métodos ***public***, ***protected*** o ***package*** de la super-clase.

Desde otras clases del ***package***:

1. Desde una clase de un ***package*** se tiene acceso a todos los miembros que no sean ***private*** de las demás clases del ***package***.

Desde otras clases fuera del ***package***:

1. Los métodos y variables son accesibles si la clase es ***public*** y el miembro es ***public***.
2. También son accesibles si la clase que accede es una ***sub-clase*** y el miembro es ***protected***.

La Tabla 3.1 muestra un resumen de los permisos de acceso en ***Java***.

Visibilidad	public	protected	private	default
Desde la propia clase	Sí	Sí	Sí	Sí
Desde otra clase en el propio package	Sí	Sí	No	Sí
Desde otra clase fuera del package	Sí	No	No	No
Desde una sub-clase en el propio package	Sí	Sí	No	Sí
Desde una sub-clase fuera del propio package	Sí	Sí	No	No

Tabla 3.1. Resumen de los permisos de acceso de Java.

3.12 TRANSFORMACIONES DE TIPO: CASTING

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

3.12.1 Conversión de tipos primitivos

La conversión entre tipos primitivos es más sencilla. En ***Java*** se realizan de modo automático conversiones implícitas ***de un tipo a otro de más precisión***, por ejemplo de ***int*** a ***long***, de ***float*** a ***double***, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto (más amplio) que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son ***conversiones inseguras*** que pueden dar lugar a errores (por ejemplo, para pasar a ***short*** un número almacenado como ***int***, hay que estar seguro de que puede ser representado con el número de cifras binarias de ***short***). A estas conversiones explícitas de tipo se les llama ***cast***. El ***cast*** se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

A diferencia de C/C++, en ***Java*** no se puede convertir un tipo numérico a ***boolean***.

La conversión de ***Strings*** (texto) a números se verá en el Apartado 4.3, en la página 69.

3.13 POLIMORFISMO

Ya se vio en el ejemplo presentado en el Apartado 1.3.8 y en los comentarios incluidos en qué consistía el ***polimorfismo***.

El ***polimorfismo*** tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama ***vinculación*** (*binding*). La ***vinculación*** puede ser ***temprana*** (en tiempo de compilación) o ***tardía*** (en tiempo de ejecución). Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en ***Java*** se utiliza siempre ***vinculación tardía***, excepto si el método es ***final***. El ***polimorfismo*** es la ***opción por defecto*** en ***Java***.

La ***vinculación tardía*** hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea ***el tipo de objeto*** y ***no el tipo de la referencia*** lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el ***polimorfismo*** necesita evaluación tardía.

El ***polimorfismo*** permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

El ***polimorfismo*** puede hacerse con referencias de ***super-clases abstract***, ***super-clases normales*** e ***interfaces***. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las ***interfaces*** permiten ampliar muchísimo las posibilidades del polimorfismo.

3.13.1 Conversión de objetos

El ***polimorfismo*** visto previamente está basado en utilizar referencias de un tipo más "amplio" que los objetos a los que apuntan. Las ventajas del polimorfismo son evidentes, pero hay una importante limitación: el tipo de la referencia (clase abstracta, clase base o interface) limita los métodos que se pueden utilizar y las variables miembro a las que se pueden acceder. Por ejemplo, un objeto puede tener una referencia cuyo tipo sea una ***interface***, aunque sólo en el caso en que su ***clase*** o ***una de sus super-clases*** implemente dicha ***interface***. Un objeto cuya referencia es un tipo ***interface*** sólo puede utilizar los métodos definidos en dicha ***interface***. Dicho de otro modo, ese objeto no puede utilizar las variables y los métodos propios de su clase. De esta forma las ***referencias de tipo interface*** definen, limitan y unifican la forma de utilizarse de objetos pertenecientes a clases muy distintas (que implementan dicha interface).

Si se desea utilizar todos los métodos y acceder a todas las variables que la clase de un objeto permite, hay que utilizar un ***cast explícito***, que convierta su referencia más general en la del tipo específico del objeto. De aquí una parte importante del interés del ***cast*** entre objetos (más bien entre referencias, habría que decir).

Para la conversión entre objetos de distintas clases, ***Java*** exige que dichas clases estén relacionadas por ***herencia*** (una deberá ser ***sub-clase*** de la otra). Se realiza una ***conversión implícita*** o ***automática*** de una ***sub-clase*** a una ***super-clase*** siempre que se necesite, ya que el objeto de la ***sub-clase*** siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la ***super-clase***. No importa que la ***super-clase*** no sea capaz de contener toda la información de la ***sub-clase***.

La conversión en sentido contrario -utilizar un objeto de una ***super-clase*** donde se espera encontrar uno de la ***sub-clase***- debe hacerse de modo ***explícito*** y puede producir errores por falta de información o de métodos. Si falta información, se obtiene una ***ClassCastException***.

No se puede acceder a las variables exclusivas de la sub-clase a través de una referencia de la super-clase. Sólo se pueden utilizar los métodos definidos en la super-clase, aunque la definición utilizada para dichos métodos sea la de la sub-clase.

Por ejemplo, supóngase que se crea un objeto de una ***sub-clase B*** y se referencia con un nombre de una ***super-clase A***,

```
A a = new B();
```

en este caso el objeto creado dispone de más información de la que la referencia ***a*** le permite acceder (podría ser, por ejemplo, una nueva variable miembro ***j*** declarada en ***B***). Para acceder a esta información adicional hay que hacer un ***cast*** explícito en la forma ***(B)a***. Para imprimir esa variable ***j*** habría que escribir (los paréntesis son necesarios):

```
System.out.println( ((B)a).j );
```

Un ***cast*** de un objeto a la ***super-clase*** puede permitir utilizar variables -no métodos- de la ***super-clase***, aunque estén redefinidos en la ***sub-clase***. Considérese el siguiente ejemplo: La clase ***C*** deriva de ***B*** y ***B*** deriva de ***A***. Las tres definen una variable ***x***. En este caso, si desde el código de la sub-clase ***C*** se utiliza:

```
x               // se accede a la x de C
this.x          // se accede a la x de C
super.x         // se accede a la x de B. Sólo se puede subir un nivel
((B)this).x     // se accede a la x de B
((A)this).x     // se accede a la x de A
```

4. CLASES DE UTILIDAD

Programando en ***Java*** nunca se parte de cero: siempre se parte de la infraestructura definida por el API de ***Java***, cuyos ***packages*** proporcionan una buena base para que el programador construya sus aplicaciones. En este Capítulo se describen algunas clases que serán de utilidad para muchos programadores.

4.1 ARRAYS

Los ***arrays*** de ***Java*** (vectores, matrices, hiper-matrices de más de dos dimensiones) se tratan como objetos de una clase predefinida. Los ***arrays*** son ***objetos***, pero con algunas características propias. Los ***arrays*** pueden ser asignados a objetos de la clase ***Object*** y los métodos de ***Object*** pueden ser utilizados con ***arrays***.

Algunas de sus características más importantes de los ***arrays*** son las siguientes:

1. Los ***arrays*** se crean con el operador ***new*** seguido del tipo y número de elementos.
2. Se puede acceder al número de elementos de un array con la variable miembro implícita ***length*** (por ejemplo, ***vect.length***).
3. Se accede a los elementos de un ***array*** con los ***corchetes []*** y un ***índice*** que varía de 0 a ***length-1***.
4. Se pueden crear ***arrays*** de objetos de cualquier tipo. En principio un ***array*** de objetos es un ***array de referencias*** que hay que completar llamando al operador ***new***.
5. Los elementos de un ***array*** se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, el carácter nulo para ***char***, ***false*** para ***boolean***, ***null*** para ***Strings*** y para referencias).
6. Como todos los objetos, los ***arrays*** se pasan como argumentos a los métodos ***por referencia***.
7. Se pueden crear ***arrays anónimos*** (por ejemplo, crear un nuevo array como argumento actual en la llamada a un método).

Inicialización de ***arrays***:

1. Los ***arrays*** se pueden inicializar con valores entre llaves {...} separados por comas.
2. También los ***arrays de objetos*** se pueden inicializar con varias llamadas a ***new*** dentro de unas llaves {...}.
3. Si se igualan dos referencias a un array no se copia el array, sino que se tiene un array con dos nombres, apuntando al mismo y único objeto.
4. Creación de una ***referencia*** a un array. Son posibles dos formas:

```
double[] x; // preferible
double x[];
```

5. Creación del ***array*** con el operador ***new***:

```
x = new double[100];
```

6. Las dos etapas 4 y 5 se pueden unir en una sola:

```
double[] x = new double[100];
```

A continuación se presentan algunos ejemplos de creación de arrays:

```
// crear un array de 10 enteros, que por defecto se inicializan a cero
int v[] = new int[10];
// crear arrays inicializando con determinados valores
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String dias[] = {"lunes", "martes", "miercoles", "jueves",
                 "viernes", "sabado", "domingo"};
// array de 5 objetos
MiClase listaObj[] = new MiClase[5]; // de momento hay 5 referencias a null
for( i = 0 ; i < 5;i++)
    listaObj[i] = new MiClase(...);
// array anónimo
obj.metodo(new String[]={"uno", "dos", "tres"});
```

4.1.1 Arrays bidimensionales

Los arrays bidimensionales de ***Java*** se crean de un modo muy similar al de C++ (con reserva dinámica de memoria). En ***Java*** una ***matriz*** es un ***vector*** de ***vectores fila***, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la ***referencia*** indicando con un doble corchete que es una ***referencia a matriz***,

   ```
   int[][] mat;
   ```

2. Crear el vector de referencias a las filas,

   ```
   mat = new int[nfilas][];
   ```

3. Reservar memoria para los vectores correspondientes a las filas,

   ```
   for (int i=0; i<nfilas; i++);
       mat[i] = new int[ncols];
   ```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
// crear una matriz 3x3
//     se inicializan a cero
double mat[][] = new double[3][3];
int [][] b = {{1, 2, 3},
              {4, 5, 6}, // esta coma es permitida
              };
int c = new[3][]; // se crea el array de referencias a arrays
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[8];
```

En el caso de una matriz ***b***, ***b.length*** es el número de filas y ***b[0].length*** es el número de columnas (de la fila 0). Por supuesto, los arrays bidimensionales pueden contener tipos primitivos de cualquier tipo u objetos de cualquier clase.

4.2 CLASES STRING Y STRINGBUFFER

Las clases ***String*** y ***StringBuffer*** están orientadas a manejar cadenas de caracteres. La clase ***String*** está orientada a manejar cadenas de caracteres constantes, es decir, que no pueden cambiar. La clase ***StringBuffer*** permite que el programador cambie la cadena insertando, borrando, etc. La primera es más eficiente, mientras que la segunda permite más posibilidades.

Ambas clases pertenecen al package ***java.lang***, y por lo tanto no hay que importarlas. Hay que indicar que el ***operador de concatenación*** (+) entre objetos de tipo ***String*** utiliza internamente objetos de la clase ***StringBuffer*** y el método ***append()***.

Los métodos de ***String*** se pueden utilizar directamente sobre ***literals*** (cadenas entre comillas), como por ejemplo: *"Hola".length()*.

4.2.1 Métodos de la clase String

Los objetos de la clase ***String*** se pueden crear a partir de cadenas constantes o *literals*, definidas entre dobles comillas, como por ejemplo: "Hola". ***Java*** crea siempre un objeto ***String*** al encontrar una cadena entre comillas. A continuación se describen dos formas de crear objetos de la clase ***String***,

```
String str1 = "Hola";             // el sistema más eficaz de crear Strings
String str2 = new String("Hola"); // también se pueden crear con un constructor
```

El primero de los métodos expuestos es el más eficiente, porque como al encontrar un texto entre comillas se crea automáticamente un objeto ***String***, en la práctica utilizando ***new*** se llama al constructor dos veces. También se pueden crear objetos de la clase ***String*** llamando a otros constructores de la clase, a partir de objetos ***StringBuffer***, y de arrays de ***bytes*** o de ***chars***.

La Tabla 4.1 muestra los métodos más importantes de la clase ***String***.

Métodos de String	Función que realizan
String(...)	Constructores para crear Strings a partir de arrays de bytes o de caracteres (ver documentación on-line)
String(String str) y String(StringBuffer sb)	Costructores a partir de un objeto String o StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
indexOf(String, [int])	Devuelve la posición en la que aparece por primera vez un String en otro String, a partir de una posición dada (opcional)
lastIndexOf(String, [int])	Devuelve la última vez que un String aparece en otro empezando en una posición y hacia el principio
length()	Devuelve el número de caracteres de la cadena
replace(char, char)	Sustituye un carácter por otro en un String
startsWith(String)	Indica si un String comienza con otro String o no
substring(int, int)	Devuelve un String extraído de otro
toLowerCase()	Convierte en minúsculas (puede tener en cuenta el locale)
toUpperCase()	Convierte en mayúsculas (puede tener en cuenta el locale)
trim()	Elimina los espacios en blanco al comienzo y final de la cadena
valueOf()	Devuelve la representación como String de sus argumento. Admite Object, arrays de caracteres y los tipos primitivos

Tabla 4.1. Algunos métodos de la clase String.

Un punto importante a tener en cuenta es que hay métodos, tales como ***System.out.println()***, que exigen que su argumento sea un objeto de la clase ***String***. Si no lo es, habrá que utilizar algún metodo que lo convierta en ***String***.

El ***locale***, citado en la Tabla 4.1, es la forma que java tiene para adaptarse a las peculiaridades de los idiomas distintos del inglés: acentos, caracteres especiales, forma de escribir las fechas y las horas, unidades monetarias, etc.

4.2.2 Métodos de la clase StringBuffer

La clase ***StringBuffer*** se utiliza prácticamente siempre que se desee modificar una cadena de caracteres. Completa los métodos de la clase ***String*** ya que éstos realizan sólo operaciones sobre el texto que no conllevan un aumento o disminución del número de letras del ***String***.

Recuérdese que hay muchos métodos cuyos argumentos deben ser objetos ***String***, que antes de pasar esos argumentos habrá que realizar la conversión correspondiente. La Tabla 4.2 muestra los métodos más importantes de la clase ***StringBuffer***.

Métodos de StringBuffer	**Función que realizan**
StringBuffer(), StringBuffer(int), StringBuffer(String)	Constructores
append(...)	Tiene muchas definiciones diferentes para añadir un String o una variable (int, long, double, etc.) a su objeto
capacity()	Devuelve el espacio libre del StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
insert(int,)	Inserta un String o un valor (int, long, double, ...) en la posición especificada de un StringBuffer
length()	Devuelve el número de caracteres de la cadena
reverse()	Cambia el orden de los caracteres
setCharAt(int, char)	Cambia el carácter en la posición indicada
setLength(int)	Cambia el tamaño de un StringBuffer
toString()	Convierte en objeto de tipo String

Tabla 4.2. Algunos métodos de la clase StringBuffer.

4.3 WRAPPERS

Los ***Wrappers*** (*envoltorios*) son clases diseñadas para ser un ***complemento*** de los ***tipos primitivos***. En efecto, los tipos primitivos son los únicos elementos de ***Java*** que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la ***eficiencia***, pero algunos inconvenientes desde el punto de vista de la ***funcionalidad***. Por ejemplo, los ***tipos primitivos*** siempre se pasan como argumento a los métodos ***por valor***, mientras que los ***objetos*** se pasan ***por referencia***. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmita al entorno que hizo la llamada. Una forma de conseguir esto es utilizar un ***Wrapper***, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases ***Wrapper*** también proporcionan métodos para realizar otras tareas con lo tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase ***Wrapper*** para cada uno de los tipos primitivos numéricos, esto es, existen las clases ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double*** (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de ***Java***). A continuación se van a ver dos de estas clases: ***Double*** e ***Integer***. Las otras cuatro son similares y sus características pueden consultarse en la documentación on-line.

4.3.1 Clase Double

La clase ***java.lang.Double*** deriva de ***Number***, que a su vez deriva de ***Object***. Esta clase contiene un valor primitivo de tipo ***double***. La Tabla 4.3 muestra algunos métodos y constantes predefinidas de la clase ***Double***.

Métodos	Función que desempeñan
Double(double) y Double(String)	Los constructores de esta clase
doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()	Métodos para obtener el valor del tipo primitivo
String toString(), Double valueOf(String)	Conversores con la clase String
isInfinite(), isNaN()	Métodos de chequear condiiciones
equals(Object)	Compara con otro objeto
MAX_DOUBLE, MIN_DOUBLE, POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN, TYPE	Constantes predefinidas. TYPE es el objeto Class representando esta clase

Tabla 4.3. Métodos y constantes de la clase Double.

El *Wrapper* ***Float*** es similar al *Wrapper* ***Double***.

4.3.2 Clase Integer

La clase ***java.lang.Integer*** tiene como variable miembro un valor de tipo ***int***. La Tabla 4.4 muestra los métodos y constantes de la clase ***Integer***.

Métodos	Función que desempeñan
Integer(int) y Integer(String)	Constructores de la clase
doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()	Conversores con otros tipos primitivos
Integer decode(String), Integer parseInt(String), String toString(), Integer valueOf(String)	Conversores con String
String toBinaryString(int), String toHexString(int), String toOctalString(int)	Conversores a cadenas representando enteros en otros sistemas de numeración
Integer getInteger(String)	Determina el valor de una propiedad del sistema a partir del nombre de dicha propiedad
MAX_VALUE, MIN_VALUE, TYPE	Constantes predefinidas

Tabla 4.4. Métodos y constantes de la clase Integer.

Los *Wrappers* ***Byte***, ***Short*** y ***Long*** son similares a ***Integer***.

Los *Wrappers* son utilizados para convertir cadenas de caracteres (texto) en números. Esto es útil cuando se leen valores desde el teclado, desde un fichero de texto, etc. Los ejemplos siguientes muestran algunas conversiones:

```
String numDecimalString = "8.978";
float  numFloat=Float.valueOf(numDecimalString).floatValue();   // numFloat = 8,979
double numDouble=Double.valueOf(numDecimalString).doubleValue();// numDouble = 8,979
String numIntString = "1001";
int    numInt=Integer.valueOf(numIntString).intValue();         // numInt = 1001
```

En el caso de que el texto no se pueda convertir directamente al tipo especificado se lanza una excepción de tipo ***NumberFormatException***, por ejemplo si se intenta convertir directamente el texto “4.897” a un número entero. El proceso que habrá que seguir será convertirlo en primer lugar a un número ***float*** y posteriormente a número entero.

4.4 CLASE MATH

La clase ***java.lang.Math*** deriva de ***Object***. La clase ***Math*** proporciona métodos ***static*** para realizar las operaciones matemáticas más habituales. Proporciona además las constantes ***E*** y ***PI***, cuyo significado no requiere muchas explicaciones.

La Tabla 4.5 muestra los métodos matemáticos soportados por esta clase.

Métodos	**Significado**	**Métodos**	**Significado**
abs()	Valor absoluto	sin(double)	Calcula el seno
acos()	Arcocoseno	tan(double)	Calcula la tangente
asin()	Arcoseno	exp()	Calcula la función exponencial
atan()	Arcotangente entre -PI/2 y PI/2	log()	Calcula el logaritmo natural (base e)
atan2(,)	Arcotangente entre -PI y PI	max(,)	Máximo de dos argumentos
ceil()	Entero más cercano en dirección a infinito	min(,)	Mínimo de dos argumentos
floor()	Entero más cercano en dirección a -infinito	random()	Número aleatorio entre 0.0 y 1.0
round()	Entero más cercano al argumento	power(,)	Devuelve el primer argumento elevado al segundo
rint(double)	Devuelve el entero más próximo	sqrt()	Devuelve la raíz cuadrada
IEEEremainder(double , double)	Calcula el resto de la división	toDegrees(double)	Pasa de radianes a grados (Java 2)
cos(double)	Calcula el coseno	toRadians()	Pasa de grados a radianes (Java 2)

Tabla 4.5. Métodos matemáticos de la clase Math.

4.5 COLECCIONES

Java dispone también de clases e interfaces para trabajar con colecciones de objetos. En primer lugar se verán las clases ***Vector*** y ***Hashtable***, así como la interface ***Enumeration***. Estas clases están presentes en lenguaje desde la primera versión. Después se explicará brevemente la ***Java Collections Framework***, introducida en la versión JDK 1.2.

4.5.1 Clase Vector

La clase ***java.util.Vector*** deriva de ***Object***, implementa ***Cloneable*** (para poder sacar copias con el método ***clone()***) y ***Serializable*** (para poder ser convertida en cadena de caracteres).

Como su mismo nombre sugiere, ***Vector*** representa un ***array de objetos*** (referencias a objetos de tipo ***Object***) que puede crecer y reducirse, según el número de elementos. Además permite acceder a los elementos con un ***índice***, aunque no permite utilizar los corchetes [].

El método ***capacity()*** devuelve el tamaño o número de elementos que puede tener el vector. El método ***size()*** devuelve el número de elementos que realmente contiene, mientras que ***capacityIncrement*** es una variable que indica el salto que se dará en el tamaño cuando se necesite crecer. La Tabla 4.6 muestra los métodos más importantes de la clase ***Vector***. Puede verse que el gran número de métodos que existen proporciona una notable flexibilidad en la utilización de esta clase.

Además de ***capacityIncrement***, existen otras dos variables miembro: ***elementCount***, que representa el número de componentes válidos del vector, y ***elementData[]*** que es el array de ***Objects*** donde realmente se guardan los elementos del objeto ***Vector*** (***capacity*** es el tamaño de este array). Las tres variables citadas son ***protected***.

Métodos	**Función que realizan**
Vector(), Vector(int), Vector(int, int)	Constructores que crean un vector vacío, un vector de la capacidad indicada y un vector de la capacidad e incremento indicados
void addElement(Object obj)	Añade un objeto al final
boolean removeElement(Object obj)	Elimina el primer objeto que encuentra como su argumento y desplaza los restantes. Si no lo encuentra devuelve false
void removeAllElements()	Elimina todos los elementos
Object clone()	Devuelve una copia del vector
void copyInto(Object anArray[])	Copia un vector en un array
void trimToSize()	Ajusta el tamaño a los elementos que tiene
void setSize(int newSize)	Establece un nuevo tamaño
int capacity()	Devuelve el tamaño (capacidad) del vector
int size()	Devuelve el número de elementos
boolean isEmpty()	Devuelve true si no tiene elementos
Enumeration elements()	Devuelve una Enumeración con los elementos
boolean contains(Object elem)	Indica si contiene o no un objeto
int indexOf(Object elem, int index)	Devuelve la posición de la primera vez que aparece un objeto a partir de una posición dada
int lastIndexOf(Object elem, int index)	Devuelve la posición de la última vez que aparece un objeto a partir de una posición, hacia atrás
Object elementAt(int index)	Devuelve el objeto en una determinada posición
Object firstElement()	Devuelve el primer elemento
Object lastElement()	Devuelve el último elemento
void setElementAt(Object obj, int index)	Cambia el elemento que está en una determinada posición
void removeElementAt(int index)	Elimina el elemento que está en una determinada posición
void insertElementAt(Object obj, int index)	Inserta un elemento por delante de una determinada posición

Tabla 4.6. Métodos de la clase Vector.

4.5.2 Interface Enumeration

La interface ***java.util.Enumeration*** define métodos útiles para recorrer una colección de objetos. Puede haber distintas clases que implementen esta interface y todas tendrán un comportamiento similar.

La interface ***Enumeration*** declara dos métodos:

1. ***public boolean hasMoreElements()***. Indica si hay más elementos en la colección o si se ha llegado ya al final.
2. ***public Object nextElement()***. Devuelve el siguiente objeto de la colección. Lanza una *NoSuchElementException* si se llama y ya no hay más elementos.

Ejemplo: Para imprimir los elementos de un vector ***vec*** se pueden utilizar las siguientes sentencias:

```
for (Enumeration e = vec.elements(); e.hasMoreElements(); ) {
    System.out.println(e.nextElement());
}
```

donde, como puede verse en la Tabla 4.6, el método ***elements()*** devuelve precisamente una referencia de tipo ***Enumeration***. Con los métodos ***hasMoreElements()*** y ***nextElement()*** y un bucle ***for*** se pueden ir imprimiendo los distintos elementos del objeto ***Vector***.

4.5.3 Clase Hashtable

La clase ***java.util.Hashtable*** extiende ***Dictionary*** (***abstract***) e implementa ***Cloneable*** y ***Serializable***. Una ***hash table*** es una tabla que relaciona una ***clave*** con un ***valor***. Cualquier objeto distinto de ***null*** puede ser tanto ***clave*** como ***valor***.

La clase a la que pertenecen las ***claves*** debe implementar los métodos ***hashCode()*** y ***equals()***, con objeto de hacer búsquedas y comparaciones. El método ***hashCode()*** devuelve un entero único y distinto para cada clave, que es siempre el mismo en una ejecución del programa pero que puede cambiar de una ejecución a otra. Además, para dos claves que resultan iguales según el método ***equals()***, el método ***hashCode()*** devuelve el mismo entero. Las ***hash tables*** están diseñadas para mantener una colección de pares ***clave/valor***, permitiendo insertar y realizar búsquedas de un modo muy eficiente

Cada objeto de ***Hashtable*** tiene dos variables: ***capacity*** y ***load factor*** (entre 0.0 y 1.0). Cuando el número de elementos excede el producto de estas variables, la ***Hashtable*** crece llamando al método ***rehash()***. Un ***load factor*** más grande apura más la memoria, pero será menos eficiente en las búsquedas. Es conveniente partir de una ***Hashtable*** suficientemente grande para no estar ampliando continuamente.

Ejemplo de definición de ***Hashtable***:

```
Hashtable numeros = new Hashtable();
numbers.put("uno", new Integer(1));
numbers.put("dos", new Integer(2));
numbers.put("tres", new Integer(3));
```

donde se ha hecho uso del método ***put()***. La Tabla 4.7 muestra los métodos de la clase ***Hashtable***.

Métodos	Función que realizan
Hashtable(), Hashtable(int nElements), Hashtable(int nElements, float loadFactor)	Constructores
int size()	Devuelve el tamaño de la tabla
boolean isEmpty()	Indica si la tabla está vacía
Enumeration keys()	Devuelve una Enumeration con las claves
Enumeration elements()	Devuelve una Enumeration con los valores
boolean contains(Object value)	Indica si hay alguna clave que se corresponde con el valor
boolean containsKey(Object key)	Indica si existe esa clave en la tabla
Object get(Object key)	Devuelve un valor dada la clave
void rehash()	Amplía la capacidad de la tabla
Object put(Object key, Object value)	Establece una relación clave-valor
Object remove(Object key)	Elimina un valor por la clave
void clear()	Limpia la tabla
Object clone()	Hace una copia de la tabla
String toString()	Devuelve un string representando la tabla

Tabla 4.7. Métodos de la clase Hashtable.

4.5.4 El Collections Framework de Java 1.2

En la versión 1.2 del JDK se introdujo el ***Java Framework Collections*** o "estructura de colecciones de Java" (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la *programación orientada a objetos*. Dada la amplitud de ***Java*** en éste y en otros aspectos se va a optar por insistir en la descripción general, dejando al lector la tarea de buscar las características concretas de los distintos métodos en la documentación de ***Java***. En este apartado se va a utilizar una forma -más breve que las tablas utilizadas en otros apartados- de informar sobre los métodos disponibles en una clase o interface.

La Figura 4.1 muestra la jerarquía de interfaces de la ***Java Collection Framework*** (JCF). En letra cursiva se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface ***Map***: ***HashMap*** y ***Hashtable***.

Las clases vistas en los apartados anteriores son clases "históricas", es decir, clases que existían antes de la versión JDK 1.2. Dichas clases se denotan en la Figura 4.1 con la letra "h" entre paréntesis. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF; en la medida de lo posible se recomienda utilizar las nuevas clases.

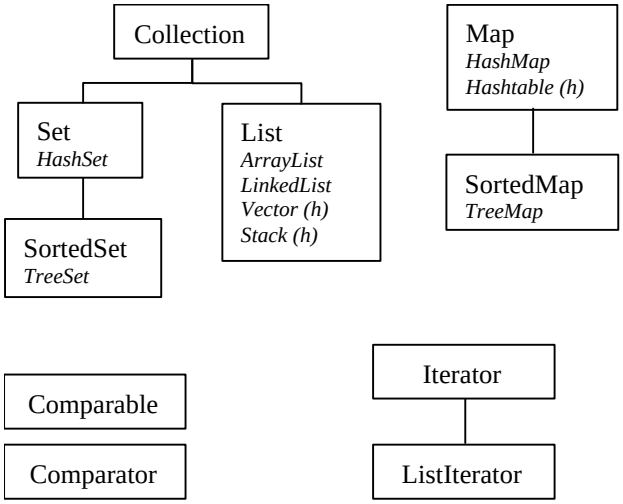


Figura 4.1. Interfaces de la Collection Framework.

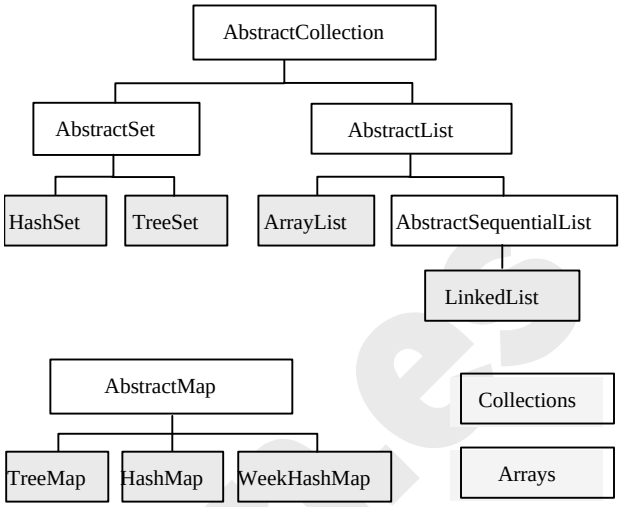


Figura 4.2. Jerarquía de clases de la Collection Framework.

En el diseño de la JCF las ***interfaces*** son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interface se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases ***ArrayList*** y ***LinkedList*** disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que ***ArrayList*** almacena los objetos en un array, la clase ***LinkedList*** los almacena en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

La Figura 4.2 muestra la jerarquía de clases de la JCF. En este caso, la jerarquía de clases es menos importante desde el punto de vista del usuario que la jerarquía de interfaces. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos.

Las clases ***Collections*** y ***Arrays*** son un poco especiales: no son ***abstract***, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos ***static*** para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

4.5.4.1 Elementos del Java Collections Framework

Interfaces de la JCF: Constituyen el elemento central de la JCF.

- ***Collection***: define métodos para tratar una colección genérica de elementos
- ***Set***: colección que no admite elementos repetidos
- ***SortedSet***: set cuyos elementos se mantienen ordenados según el criterio establecido
- ***List***: admite elementos repetidos y mantiene un orden inicial
- ***Map***: conjunto de pares clave/valor, sin repetición de claves
- ***SortedMap***: map cuyos elementos se mantienen ordenados según el criterio establecido

Interfaces de soporte:

- ***Iterator***: sustituye a la interface ***Enumeration***. Dispone de métodos para recorrer una colección y para borrar elementos.
- ***ListIterator***: deriva de ***Iterator*** y permite recorrer lists en ambos sentidos.
- ***Comparable***: declara el método ***compareTo***() que permite ordenar las distintas colecciones según un orden natural (***String***, ***Date***, ***Integer***, ***Double***, …).
- ***Comparator***: declara el método ***compare***() y se utiliza en lugar de ***Comparable*** cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

- ***HashSet***: Interface ***Set*** implementada mediante una hash table.
- ***TreeSet***: Interface ***SortedSet*** implementada mediante un árbol binario ordenado.
- ***ArrayList***: Interface ***List*** implementada mediante un array.
- ***LinkedList***: Interface ***List*** implementada mediante una lista vinculada.
- ***HashMap***: Interface ***Map*** implementada mediante una hash table.
- ***WeekHashMap***: Interface ***Map*** implementada de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la ***WeekHashMap***.
- ***TreeMap***: Interface ***SortedMap*** implementada mediante un árbol binario

Clases Wrapper: Colecciones con características adicionales, como no poder ser modificadas o estar sincronizadas. No se accede a ellas mediante constructores, sino mediante métodos "factory" de la clase ***Collections***.

Clases de utilidad: Son mini-implementaciones que permiten obtener sets especializados, como por ejemplo ***sets*** constantes de un sólo elemento (***singleton***) o ***lists*** con ***n*** copias del mismo elemento (***nCopies***). Definen las constantes EMPTY_SET y MPTY_LIST. Se accede a través de la clase ***Collections***.

Clases históricas: Son las clases ***Vector*** y ***Hashtable*** presentes desde las primeras versiones de ***Java***. En las versiones actuales, implementan respectivamente las interfaces ***List*** y ***Map***, aunque conservan también los métodos anteriores.

Clases abstractas: Son las clases abstract de la Figura 4.2. Tienen total o parcialmente implementados los métodos de la interface correspondiente. Sirven para que los usuarios deriven de ellas sus propias clases con un mínimo de esfuerzo de programación.

Algoritmos: La clase ***Collections*** dispone de métodos ***static*** para ordenar, desordenar, invertir orden, realizar búsquedas, llenar, copiar, hallar el mínimo y hallar el máximo.

Clase Arrays: Es una clase de utilidad introducida en el JDK 1.2 que contiene métodos ***static*** para ordenar, llenar, realizar búsquedas y comparar los arrays clásicos del lenguaje. Permite también ver los ***arrays*** como ***lists***.

Después de esta visión general de la *Java Collections Framework*, se verán algunos detalles de las clases e interfaces más importantes.

4.5.4.2 Interface Collection

La interface ***Collection*** es implementada por los ***conjuntos*** (*sets*) y las ***listas*** (*lists*). Esta interface declara una serie de métodos generales utilizables con ***Sets*** y ***Lists***. La declaración o header de dichos métodos se puede ver ejecutando el comando **> *javap java.util.Collection*** en una ventana de MS-DOS. El resultado se muestra a continuación:

```
public interface java.util.Collection
{
    public abstract boolean add(java.lang.Object);              // opcional
    public abstract boolean addAll(java.util.Collection);       // opcional
    public abstract void clear();                               // opcional
    public abstract boolean contains(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Iterator iterator();
    public abstract boolean remove(java.lang.Object);           // opcional
    public abstract boolean removeAll(java.util.Collection);    // opcional
    public abstract boolean retainAll(java.util.Collection);    // opcional
    public abstract int size();
    public abstract java.lang.Object toArray()[];
    public abstract java.lang.Object toArray(java.lang.Object[])[];
}
```

A partir del nombre, de los argumentos y del valor de retorno, la mayor parte de estos métodos resultan autoexplicativos. A continuación se introducen algunos comentarios sobre los aspectos que pueden resultar más novedosos de estos métodos. Los detalles se pueden consultar en la documentación de ***Java***.

Los métodos indicados como "// opcional" (estos caracteres han sido introducidos por los autores de este manual) no están disponibles en algunas implementaciones, como por ejemplo en las clases que no permiten modificar sus objetos. Por supuesto, dichos métodos deben ser definidos, pero lo que hacen al ser llamados es lanzar una ***UnsupportedOperationException***.

El método ***add()*** trata de añadir un objeto a una colección, pero puede que no lo consiga si la colección es un ***set*** que ya tiene ese elemento. Devuelve ***true*** si el método ha llegado a modificar la colección. Lo mismo sucede con ***addAll()***. El método ***remove()*** elimina un único elemento (si lo encuentra); devuelve ***true*** si la colección ha sido modificada.

El método ***iterator()*** devuelve una referencia ***Iterator*** que permite recorrer una colección con los métodos ***next()*** y ***hasNext()***. Permite también borrar el elemento actual con ***remove()***.

Los dos métodos ***toArray()*** permiten convertir una colección en un array.

4.5.4.3 Interfaces Iterator y ListIterator

La interface ***Iterator*** sustituye a ***Enumeration***, utilizada en versiones anteriores del JDK. Dispone de los métodos siguientes:

```
Compiled from Iterator.java
public interface java.util.Iterator
{
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}
```

El método ***remove()*** permite borrar el último elemento accedido con ***next()***. Es la única forma segura de eliminar un elemento mientras se está recorriendo una colección.

Los métodos de la interface ***ListIterator*** son los siguientes:

```
Compiled from ListIterator.java
public interface java.util.ListIterator extends java.util.Iterator
{
    public abstract void add(java.lang.Object);
    public abstract boolean hasNext();
    public abstract boolean hasPrevious();
    public abstract java.lang.Object next();
    public abstract int nextIndex();
    public abstract java.lang.Object previous();
    public abstract int previousIndex();
    public abstract void remove();
    public abstract void set(java.lang.Object);
}
```

La interface ***ListIterator*** permite recorrer una lista en ambas direcciones, y hacer algunas modificaciones mientras se recorre. Los elementos se numeran desde 0 a *n-1*, pero los valores válidos para el índice son de 0 a *n*. Puede suponerse que el índice *i* está en la frontera entre los elementos *i-1* e *i*: en ese caso ***previousIndex***() devolvería *i-1* y ***nextIndex()*** devolvería *i*. Si el índice es 0, ***previousIndex()*** devuelve –1 y si el índice es *n* ***nextIndex()*** devuelve el resultado de ***size()***.

4.5.4.4 Interfaces Comparable y Comparator

Estas interfaces están orientadas a mantener ordenadas las ***listas***, y también los ***sets*** y ***maps*** que deben mantener un orden. Para ello se dispone de las interfaces ***java.lang.Comparable*** y ***java.util.Comparator*** (obsérvese que pertenecen a packages diferentes).

La interface ***Comparable*** declara el método ***compareTo()*** de la siguiente forma:

```
public int compareTo(Object obj)
```

que compara su argumento implícito con el que se le pasa por ventana. Este método devuelve un entero ***negativo***, ***cero*** o ***positivo*** según el argumento implícito (***this***) sea ***anterior***, ***igual*** o ***posterior*** al objeto ***obj***. Las listas de objetos de clases que implementan esta interface tienen un ***orden natural***. En ***Java 1.2*** esta interface está implementada -entre otras- por las clases ***String***, ***Character***, ***Date***, ***File***, ***BigDecimal***, ***BigInteger***, ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double***. Téngase en cuenta que la implementación estándar de estas clases no asegura un orden alfabético correcto con mayúsculas y minúsculas, y tampoco en idiomas distintos del inglés.

Si se redefine, el método ***compareTo()*** debe ser programado con cuidado: es muy conveniente que sea coherente con el método ***equals()*** y que cumpla la ***propiedad transitiva***. Para más información, consultar la documentación del JDK 1.2.

Las ***listas*** y los ***arrays*** cuyos elementos implementan ***Comparable*** pueden ser ordenadas con los métodos static ***Collections.sort()*** y ***Arrays.sort()***.

La interface ***Comparator*** permite ordenar listas y colecciones cuyos objetos pertenecen a clases de tipo cualquiera. Esta interface permitiría por ejemplo ordenar figuras geométricas planas por el área o el perímetro. Su papel es similar al de la interface ***Comparable***, pero el usuario debe siempre proporcionar una implementación de esta clase. Sus dos métodos se declaran en la forma:

```
public int compare(Object o1, Object o2)
public boolean equals(Object obj)
```

El objetivo del método ***equals()*** es comparar ***Comparators***.

El método ***compare()*** devuelve un entero ***negativo***, ***cero*** o ***positivo*** según su primer argumento sea ***anterior***, ***igual*** o ***posterior*** al segundo. Los objetos que implementan ***Comparator*** pueden pasarse como argumentos al método ***Collections.sort()*** o a algunos ***constructores*** de las clases ***TreeSet*** y ***TreeMap***, con la idea de que las mantengan ordenadas de acuerdo con dicho ***Comparator***. Es muy importante que ***compare()*** sea compatible con el método ***equals()*** de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de ***compareTo()***.

Java 1.2 dispone de clases capaces de ordenar cadenas de texto en diferentes lenguajes. Para ello se puede consultar la documentación sobre las clases ***CollationKey***, ***Collator*** y sus clases derivadas, en el package ***java.text***.

4.5.4.5 Sets y SortedSets

La interface ***Set*** sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada (con un orden natural o definido por el usuario, se entiende). La interface ***Set*** no declara ningún método adicional a los de ***Collection***.

Como un ***Set*** no admite elementos repetidos es importante saber cuándo dos objetos son considerados iguales (por ejemplo, el usuario puede o no desear que las palabras ***Mesa*** y ***mesa*** sean consideradas iguales). Para ello se dispone de los métodos ***equals()*** y ***hashcode()***, que el usuario puede redefinir si lo desea.

Utilizando los métodos de ***Collection***, los ***Sets*** permiten realizar operaciones algebraicas de ***unión***, ***intersección*** y ***diferencia***. Por ejemplo, ***s1.containsAll(s2)*** permite saber si ***s2*** está contenido en ***s1***; ***s1.addAll(s2)*** permite convertir ***s1*** en la unión de los dos conjuntos; ***s1.retainAll(s2)*** permite convertir ***s1*** en la intersección de ***s1*** y ***s2***; finalmente, ***s1.removeAll(s2)*** convierte ***s1*** en la diferencia entre ***s1*** y ***s2***.

La interface ***SortedSet*** extiende la interface ***Set*** y añade los siguientes métodos:

```
Compiled from SortedSet.java
public interface java.util.SortedSet extends java.util.Set
{
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object first();
    public abstract java.util.SortedSet headSet(java.lang.Object);
    public abstract java.lang.Object last();
    public abstract java.util.SortedSet subSet(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedSet tailSet(java.lang.Object);
}
```

que están orientados a trabajar con el "orden". El método ***comparator()*** permite obtener el objeto pasado al constructor para establecer el orden. Si se ha utilizado el orden natural definido por la interface ***Comparable***, este método devuelve ***null***. Los métodos ***first()*** y ***last()*** devuelven el primer y último elemento del conjunto. Los métodos ***headSet()***, ***subSet()*** y ***tailSet()*** sirven para obtener sub-conjuntos al principio, en medio y al final del conjunto original (los dos primeros no incluyen el límite superior especificado).

Existes dos implementaciones de conjuntos: la clase ***HashSet*** implementa la interface ***Set***, mientras que la clase ***TreeSet*** implementa ***SortedSet***. La primera está basada en una hash table y la segunda en un ***TreeMap***.

Los elementos de un ***HashSet*** no mantienen el orden natural, ni el orden de introducción. Los elementos de un ***TreeSet*** mantienen el orden natural o el especificado por la interface ***Comparator***. Ambas clases definen constructores que admiten como argumento un objeto ***Collection***, lo cual permite convertir un ***HashSet*** en un ***TreeSet*** y viceversa.

4.5.4.6 Listas

La interface ***List*** define métodos para operar con colecciones ordenadas y que pueden tener elementos repetidos. Por ello, dicha interface declara métodos adicionales que tienen que ver con el orden y con el acceso a elementos o rangos de elementos. Además de los métodos de Collection, la interface ***List*** declara los métodos siguientes:

```
Compiled from List.java
public interface java.util.List extends java.util.Collection
{
    public abstract void add(int, java.lang.Object);
    public abstract boolean addAll(int, java.util.Collection);
    public abstract java.lang.Object get(int);
    public abstract int indexOf(java.lang.Object);
    public abstract int lastIndexOf(java.lang.Object);
    public abstract java.util.ListIterator listIterator();
    public abstract java.util.ListIterator listIterator(int);
    public abstract java.lang.Object remove(int);
    public abstract java.lang.Object set(int, java.lang.Object);
    public abstract java.util.List subList(int, int);
}
```

Los nuevos métodos ***add()*** y ***addAll()*** tienen un argumento adicional para insertar elementos en una posición determinada, desplazando el elemento que estaba en esa posición y los siguientes. Los métodos ***get()*** y ***set()*** permiten obtener y cambiar el elemento en una posición dada. Los métodos ***indexOf()*** y ***lastIndexOf()*** permiten saber la posición de la primera o la última vez que un elemento aparece en la lista; si el elemento no se encuentra se devuelve –1.

El método ***subList(int fromIndex, toIndex)*** devuelve una "vista" de la lista, desde el elemento ***fromIndex*** inclusive hasta el ***toIndex*** exclusive. Un cambio en esta "vista" se refleja en la lista original, aunque no conviene hacer cambios simultáneamente en ambas. Lo mejor es eliminar la "vista" cuando ya no se necesita.

Existen dos implementaciones de la interface ***List***, que son las clases ***ArrayList*** y ***LinkedList***. La diferencia está en que la primera almacena los elementos de la colección en un ***array*** de ***Objects***, mientras que la segunda los almacena en una ***lista vinculada***. Los ***arrays*** proporcionan una forma de acceder a los elementos mucho más eficiente que las listas vinculadas. Sin embargo tienen dificultades para crecer (hay que reservar memoria nueva, copiar los elementos del array antiguo y liberar la memoria) y para insertar y/o borrar elementos (hay que desplazar en un sentido u en otro los elementos que están detrás del elemento borrado o insertado). Las ***listas vinculadas*** sólo permiten acceso secuencial, pero tienen una gran flexibilidad para crecer, para borrar y para insertar elementos. El optar por una implementación u otra depende del caso concreto de que se trate.

4.5.4.7 Maps y SortedMaps

Un ***Map*** es una estructura de datos agrupados en parejas ***clave/valor***. Pueden ser considerados como una tabla de dos columnas. La ***clave*** debe ser única y se utiliza para acceder al ***valor***.

Aunque la interface ***Map*** no deriva de ***Collection***, es posible ver los ***Maps*** como colecciones de ***claves***, de ***valores*** o de parejas ***clave/valor***. A continuación se muestran los métodos de la interface ***Map*** (comando **> *javap java.util.Map***):

```
Compiled from Map.java
public interface java.util.Map
{
    public abstract void clear();
    public abstract boolean containsKey(java.lang.Object);
    public abstract boolean containsValue(java.lang.Object);
    public abstract java.util.Set entrySet();
    public abstract boolean equals(java.lang.Object);
    public abstract java.lang.Object get(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Set keySet();
    public abstract java.lang.Object put(java.lang.Object, java.lang.Object);
    public abstract void putAll(java.util.Map);
    public abstract java.lang.Object remove(java.lang.Object);
    public abstract int size();
    public abstract java.util.Collection values();
    public static interface java.util.Map.Entry
    {
        public abstract boolean equals(java.lang.Object);
        public abstract java.lang.Object getKey();
        public abstract java.lang.Object getValue();
        public abstract int hashCode();
        public abstract java.lang.Object setValue(java.lang.Object);
    }
}
```

Muchos de estos métodos tienen un significado evidente, pero otros no tanto. El método ***entrySet()*** devuelve una "vista" del ***Map*** como ***Set***. Los elementos de este ***Set*** son referencias de la interface ***Map.Entry***, que es una ***interface interna*** de ***Map***. Esta "vista" del ***Map*** como ***Set*** permite modificar y eliminar elementos del ***Map***, pero no añadir nuevos elementos.

El método ***get(key)*** permite obtener el valor a partir de la clave. El método ***keySet()*** devuelve una "vista" de las claves como ***Set***. El método ***values()*** devuelve una "vista" de los valores del ***Ma***p como ***Collection*** (porque puede haber elementos repetidos). El método ***put()*** permite añadir una pareja clave/valor, mientras que ***putAll()*** vuelca todos los elementos de un ***Map*** en otro ***Map*** (los pares con clave nueva se añaden; en los pares con clave ya existente los valores nuevos sustituyen a los antiguos). El método ***remove()*** elimina una pareja clave/valor a partir de la clave.

La interface ***SortedMap*** añade los siguientes métodos, similares a los de ***SortedSet***:

```
Compiled from SortedMap.java
public interface java.util.SortedMap extends java.util.Map
{
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object firstKey();
    public abstract java.util.SortedMap headMap(java.lang.Object);
    public abstract java.lang.Object lastKey();
    public abstract java.util.SortedMap subMap(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedMap tailMap(java.lang.Object);
}
```

La clase ***HashMap*** implementa la interface ***Map*** y está basada en una hash table, mientras que ***TreeMap*** implementa ***SortedMap*** y está basada en un árbol binario.

4.5.4.8 Algoritmos y otras características especiales: Clases Collections y Arrays

La clase ***Collections*** (no confundir con la interface ***Collection***, en singular) es una clase que define un buen número de métodos static con diversas finalidades. No se detallan o enumeran aquí porque exceden del espacio disponible. Los más interesantes son los siguientes:

- Métodos que definen algoritmos:

  Ordenación mediante el método mergesort
  ```
  public static void sort(java.util.List);
  public static void sort(java.util.List, java.util.Comparator);
  ```
  Eliminación del orden de modo aleatorio
  ```
  public static void shuffle(java.util.List);
  public static void shuffle(java.util.List, java.util.Random);
  ```
  Inversión del orden establecido
  ```
  public static void reverse(java.util.List);
  ```
  Búsqueda en una lista
  ```
  public static int binarySearch(java.util.List, java.lang.Object);
  public static int binarySearch(java.util.List, java.lang.Object,
                                 java.util.Comparator);
  ```
  Copiar una lista o reemplazar todos los elementos con el elemento especificado
  ```
  public static void copy(java.util.List, java.util.List);
  public static void fill(java.util.List, java.lang.Object);
  ```
  Cálculo de máximos y mínimos
  ```
  public static java.lang.Object max(java.util.Collection);
  public static java.lang.Object max(java.util.Collection, java.util.Comparator);
  public static java.lang.Object min(java.util.Collection);
  public static java.lang.Object min(java.util.Collection, java.util.Comparator);
  ```
- Métodos de utilidad

  Set inmutable de un único eleento
  ```
  public static java.util.Set singleton(java.lang.Object);
  ```
  Lista inmutable con n copias de un objeto
  ```
  public static java.util.List nCopies(int, java.lang.Object);
  ```
  Constantes para representar el conjunto y la lista vacía
  ```
  public static final java.util.Set EMPTY_SET;
  public static final java.util.List EMPTY_LIST;
  ```

Además, la clase ***Collections*** dispone de dos conjuntos de métodos "factory" que pueden ser utilizados para convertir objetos de distintas colecciones en objetos "***read only***" y para convertir distintas colecciones en objetos "***synchronized***" (por defecto las clases vistas anteriormente no están sincronizadas, lo cual quiere decir que se puede acceder a la colección desde distintas threads sin que se produzcan problemas. Los métodos correspondientes son los siguientes:

```
public static java.util.Collection synchronizedCollection(java.util.Collection);
public static java.util.List synchronizedList(java.util.List);
public static java.util.Map synchronizedMap(java.util.Map);
public static java.util.Set synchronizedSet(java.util.Set);
public static java.util.SortedMap synchronizedSortedMap(java.util.SortedMap);
public static java.util.SortedSet synchronizedSortedSet(java.util.SortedSet);

public static java.util.Collection unmodifiableCollection(java.util.Collection);
public static java.util.List unmodifiableList(java.util.List);
public static java.util.Map unmodifiableMap(java.util.Map);
public static java.util.Set unmodifiableSet(java.util.Set);
public static java.util.SortedMap unmodifiableSortedMap(java.util.SortedMap);
```

```
public static java.util.SortedSet unmodifiableSortedSet(java.util.SortedSet);
```

Estos método se utilizan de una forma muy sencilla: se les pasa como argumento una referencia a un objeto que no cumple la característica deseada y se obtiene como valor de retorno una referencia a un objeto que sí la cumple.

4.5.4.9 Desarrollo de clases por el usuario: clases abstract

Las clases abstract indicadas en la Figura 4.2, en la página 75, pueden servir como base para que los programadores con necesidades no cubiertas por las clases vistas anteriormente desarrollen sus propias clases.

4.5.4.10 Interfaces Cloneable y Serializable

Las clases ***HashSet***, ***TreeSet***, ***ArrayList***, ***LinkedList***, ***HashMap*** y ***TreeMap*** (al igual que ***Vector*** y ***Hashtable***) implementan las interfaces ***Cloneable*** y ***Serializable***, lo cual quiere decir que es correcto sacar copias bit a bit de sus objetos con el método ***Object.clone()***, y que se pueden convertir en cadenas o flujos (streams) de caracteres.

Una de las ventajas de implementar la interface ***Serializable*** es que los objetos de estas clases pueden ser impresos con los métodos ***System.Out.print()*** y ***System.Out.println()***.

4.6 OTRAS CLASES DEL PACKAGE JAVA.UTIL

El package ***java.util*** tiene otras clases interesantes para aplicaciones de distinto tipo, entre ellas algunas destinadas a considerar todo lo relacionado con fechas y horas. A continuación se consideran algunas de dichas clases.

4.6.1 Clase Date

La clase ***Date*** representa un instante de tiempo dado con precisión de milisegundos. La información sobre fecha y hora se almacena en un entero long de 64 bits que contiene los milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970 GMT (*Greenwich mean time*). Ya se verá que otras clases permiten a partir de un objeto ***Date*** obtener información del año, mes, día, horas, minutos y segundos. A continuación se muestran los métodos de la clase ***Date***, habiéndose eliminado los métodos declarados obsoletos (*deprecated*) en el JDK 1.2:

```
Compiled from Date.java
public class java.util.Date extends java.lang.Object implements
             java.io.Serializable, java.lang.Cloneable, java.lang.Comparable {
    public java.util.Date();
    public java.util.Date(long);
    public boolean after(java.util.Date);
    public boolean before(java.util.Date);
    public java.lang.Object clone();
    public int compareTo(java.lang.Object);
    public int compareTo(java.util.Date);
    public boolean equals(java.lang.Object);
    public long getTime();
    public int hashCode();
    public void setTime(long);
    public java.lang.String toString();
}
```

El constructor por defecto ***Date()*** crea un objeto a partir de la fecha y hora actual del ordenador. El segundo constructor crea el objeto a partir de los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT. Los métodos ***after()*** y ***before()*** permiten saber si la fecha indicada como argumento implícito (***this***) es posterior o anterior a la pasada como argumento. Los métodos ***getTime()*** y ***setTime()*** permiten obtener o establecer los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT para un determinado objeto ***Date***. Otros métodos son consecuencia de las interfaces implementadas por la clase ***Date***.

Los objetos de esta clase no se utilizan mucho directamente, sino que se utilizan en combinación con las clases que se vana ver a continuación.

4.6.2 Clases Calendar y GregorianCalendar

La clase ***Calendar*** es una clase ***abstract*** que dispone de métodos para convertir objetos de la clase ***Date*** en enteros que representan fechas y horas concretas. La clase ***GregorianCalendar*** es la única clase que deriva de ***Calendar*** y es la que se utilizará de ordinario.

Java tiene una forma un poco particular para representar las fechas y horas:

1. Las horas se representan por enteros de 0 a 23 (la hora o va de las 00:00:00 hasta la 1:00:00), y los minutos y segundos por enteros entre 0 y 59.
2. Los días del mes se representan por enteros entre 1 y 31 (lógico).
3. Los meses del año se representan mediante enteros de 0 a 11 (no tan lógico).
4. Los años se representan mediante enteros de cuatro dígitos. Si se representan con dos dígitos, se resta 1900. Por ejemplo, con dos dígitos el año 2000 es para Java el año 00.

La clase ***Calendar*** tiene una serie de variables miembro y constantes (variables ***final***) que pueden resultar muy útiles:

- La variable ***int*** AM_PM puede tomar dos valores: las constantes enteras AM y PM.
- La variable ***int*** DAY_OF_WEEK puede tomar los valores ***int*** SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY y SATURDAY.
- La variable ***int*** MONTH puede tomar los valores ***int*** JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER. Para hacer los programas más legibles es preferible utilizar estas constantes simbólicas que los correspondientes números del 0 al 11.
- La variable miembro HOUR se utiliza en los métodos get() y set() para indicar la hora de la mañana o de la tarde (en relojes de 12 horas, de 0 a 11). La variable HOUR_OF_DAY sirve para indicar la hora del día en relojes de 24 horas (de 0 a 23).
- Las variables DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_MONTH (o bien DATE), DAY_OF_YEAR, WEEK_OF_MONTH, WEEK_OF_YEAR tienen un significado evidente.
- Las variables ERA, YEAR, MONTH, HOUR, MINUTE, SECOND, MILLISECOND tienen también un significado evidente.

- Las variables ZONE_OFFSET y DST_OFFSET indican la zona horaria y el desafíe en milisegundos respecto a la zona GMT.

La clase ***Calendar*** dispone de un gran número de métodos para establecer u obtener los distintos valores de la fecha y/u hora. Algunos de ellos se muestran a continuación. Para más información, se recomienda utilizar la documentación de JDK 1.2.

```
Compiled from Calendar.java
public abstract class java.util.Calendar extends java.lang.Object implements
java.io.Serializable, java.lang.Cloneable {
    protected long time;
    protected boolean isTimeSet;
    protected java.util.Calendar();
    protected java.util.Calendar(java.util.TimeZone,java.util.Locale);
    public abstract void add(int, int);
    public boolean after(java.lang.Object);
    public boolean before(java.lang.Object);
    public final void clear();
    public final void clear(int);
    protected abstract void computeTime();
    public boolean equals(java.lang.Object);
    public final int get(int);
    public int getFirstDayOfWeek();
    public static synchronized java.util.Calendar getInstance();
    public static synchronized java.util.Calendar getInstance(java.util.Locale);
    public static synchronized java.util.Calendar getInstance(java.util.TimeZone);
    public static synchronized java.util.Calendar getInstance(java.util.TimeZone,
                                                              java.util.Locale);
    public final java.util.Date getTime();
    protected long getTimeInMillis();
    public java.util.TimeZone getTimeZone();
    public final boolean isSet(int);
    public void roll(int, int);
    public abstract void roll(int, boolean);
    public final void set(int, int);
    public final void set(int, int, int);
    public final void set(int, int, int, int, int);
    public final void set(int, int, int, int, int, int);
    public final void setTime(java.util.Date);
    public void setFirstDayOfWeek(int);
    protected void setTimeInMillis(long);
    public void setTimeZone(java.util.TimeZone);
    public java.lang.String toString();
}
```

La clase ***GregorianCalendar*** añade las constante BC y AD para la ERA, que representan respectivamente antes y después de Jesucristo. Añade además varios constructores que admiten como argumentos la información correspondiente a la fecha/hora y –opcionalmente- la zona horaria.

A continuación se muestra un ejemplo de utilización de estas clases. Se sugiere al lector que cree y ejecute el siguiente programa, observando los resultados impresos en la consola.

```
import java.util.*;

public class PruebaFechas {
    public static void main(String arg[]) {
        Date d = new Date();
        GregorianCalendar gc = new GregorianCalendar();
        gc.setTime(d);
        System.out.println("Era:            "+gc.get(Calendar.ERA));
        System.out.println("Year:           "+gc.get(Calendar.YEAR));
        System.out.println("Month:          "+gc.get(Calendar.MONTH));
        System.out.println("Dia del mes:    "+gc.get(Calendar.DAY_OF_MONTH));
```

```
        System.out.println("D de la S en mes:"
                           +gc.get(Calendar.DAY_OF_WEEK_IN_MONTH));
        System.out.println("No de semana:    "+gc.get(Calendar.WEEK_OF_YEAR));
        System.out.println("Semana del mes:  "+gc.get(Calendar.WEEK_OF_MONTH));
        System.out.println("Fecha:           "+gc.get(Calendar.DATE));
        System.out.println("Hora:            "+gc.get(Calendar.HOUR));
        System.out.println("Tiempo del dia:  "+gc.get(Calendar.AM_PM));
        System.out.println("Hora del dia:    "+gc.get(Calendar.HOUR_OF_DAY));
        System.out.println("Minuto:          "+gc.get(Calendar.MINUTE));
        System.out.println("Segundo:         "+gc.get(Calendar.SECOND));
        System.out.println("Dif. horaria:    "+gc.get(Calendar.ZONE_OFFSET));
    }
}
```

4.6.3 Clases DateFormat y SimpleDateFormat

DateFormat es una clase ***abstract*** que pertenece al package ***java.text*** y no al package ***java.util***, como las vistas anteriormente. La razón es para facilitar todo lo referente a la internacionalización, que es un aspecto muy importante en relación con la conversión, que permite dar formato a fechas y horas de acuerdo con distintos criterios locales. Esta clase dispone de métodos ***static*** para convertir ***Strings*** representando fechas y horas en objetos de la clase ***Date***, y viceversa.

La clase ***SimpleDateFormat*** es la única clase derivada de ***DateFormat***. Es la clase que conviene utilizar. Esta clase se utiliza de la siguiente forma: se le pasa al constructor un ***String*** definiendo el formato que se desea utilizar. Por ejemplo:

```
import java.util.*;
import java.text.*;

class SimpleDateForm {
    public static void main(String arg[]) throws ParseException {
        SimpleDateFormat sdf1 = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
        SimpleDateFormat sdf2 = new SimpleDateFormat("dd-MM-yy");
        Date d = sdf1.parse("12-04-1968 11:23:45");
        String s = sdf2.format(d);
        System.out.println(s);
    }
}
```

La documentación de la clase ***SimpleDateFormat*** proporciona abundante información al respecto, incluyendo algunos ejemplos.

4.6.4 Clases TimeZone y SimpleTimeZone

La clase ***TimeZone*** es también una clase abstract que sirve para definir la zona horaria. Los métodos de esta clase son capaces de tener en cuenta el cambio de la hora en verano para ahorrar energía. La clase ***SimpleTimeZone*** deriva de ***TimeZone*** y es la que conviene utilizar.

El valor por defecto de la zona horaria es el definido en el ordenador en que se ejecuta el programa. Los objetos de esta clase pueden ser utilizados con los constructores y algunos métodos de la clase ***Calendar*** para establecer la zona horaria.

5. EL AWT (ABSTRACT WINDOWS TOOLKIT)

5.1 QUÉ ES EL AWT

El AWT (*Abstract Windows Toolkit*) es la parte de ***Java*** que se ocupa de construir interfaces gráficas de usuario. Aunque el AWT ha estado presente en ***Java*** desde la versión 1.0, la versión 1.1 representó un cambio notable, sobre todo en lo que respecta al ***modelo de eventos***. La versión 1.2 ha incorporado un modelo distinto de componentes llamado ***Swing***, que también está disponible en la versión 1.1 como package adicional. En este Capítulo se seguirá el AWT de ***Java 1.1***, también soportado por la versión 1.2.

5.1.1 Creación de una Interface Gráfica de Usuario

Para construir una interface gráfica de usuario hace falta:

1. Un "contenedor" o ***container***, que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos. Se correspondería con un ***formulario*** o una ***picture box*** de ***Visual Basic***.
2. Los ***componentes***: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc. Se corresponderían con los ***controles*** de ***Visual Basic***.
3. El ***modelo de eventos***. El usuario controla la aplicación actuando sobre los componentes, de ordinario con el ratón o con el teclado. Cada vez que el usuario realiza una determinada acción, se produce el ***evento*** correspondiente, que el sistema operativo transmite al AWT. El AWT crea un ***objeto*** de una determinada clase de evento, derivada de ***AWTEvent***. Este ***evento*** es transmitido a un determinado ***método*** para que lo gestione. En ***Visual Basic*** el entorno de desarrollo crea automáticamente el procedimiento que va a gestionar el evento (uniendo el nombre del control con el tipo del evento mediante el carácter _) y el usuario no tiene más que introducir el código. En ***Java*** esto es un poco más complicado: el componente u objeto que recibe el evento debe "registrar" o indicar previamente qué objeto se va a hacer cargo de gestionar ese evento.

En los siguientes apartados se verán con un cierto detalle estos tres aspectos del AWT. Hay que considerar que el AWT es una parte muy extensa y complicada de ***Java***, sobre la que existen libros con muchos cientos de páginas.

5.1.2 Objetos "event source" y objetos "event listener"

El modelo de eventos de ***Java*** está basado en que los objetos sobre los que se producen los eventos (***event sources***) "registran" los objetos que habrán de gestionarlos (***event listeners***), para lo cual los ***event listeners*** habrán de disponer de los ***métodos*** adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los ***event listeners*** disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface ***Listener***. Las interfaces ***Listener*** se corresponden con los tipos de ***eventos*** que se pueden producir. En los apartados siguientes se verán con más detalle los ***componentes*** que

pueden recibir ***eventos***, los distintos tipos de ***eventos*** y los ***métodos*** de las interfaces ***Listener*** que hay que definir para gestionarlos. En este punto es muy importante ser capaz de buscar la información correspondiente en la documentación de ***Java***.

Las capacidades gráficas del AWT resultan pobres y complicadas en comparación con lo que se puede conseguir con ***Visual Basic***, pero tienen la ventaja de poder ser ejecutadas casi en cualquier ordenador y con cualquier sistema operativo.

5.1.3 Proceso a seguir para crear una aplicación interactiva (orientada a eventos)

Para avanzar un paso más, se resumen a continuación los pasos que se pueden seguir para construir una aplicación orientada a eventos sencilla, con interface gráfica de usuario:

1. Determinar los ***componentes*** que van a constituir la interface de usuario (botones, cajas de texto, menús, etc.).
2. Crear una ***clase*** para la aplicación que contenga la función ***main()***.
3. Crear una clase ***Ventana***, sub-clase de ***Frame***, que responda al evento ***WindowClosing()***.
4. La función ***main()*** deberá crear un objeto de la clase ***Ventana*** (en el que se van a introducir las componentes seleccionadas) y mostrarla por pantalla con el tamaño y posición adecuados.
5. Añadir al objeto ***Ventana*** todos los ***componentes*** y ***menús*** que deba contener. Se puede hacer en el constructor de la ventana o en el propio método ***main()***.
6. Definir los objetos ***Listener*** (objetos que se ocuparán de responder a los eventos, cuyas clases implementan las distintas interfaces ***Listener***) para cada uno de los eventos que deban estar soportados. En aplicaciones pequeñas, el propio objeto ***Ventana*** se puede ocupar de responder a los eventos de sus componentes. En programas más grandes se puede crear uno o más objetos de clases especiales para ocuparse de los eventos.
7. Finalmente, se deben implementar los métodos de las interfaces ***Listener*** que se vayan a hacer cargo de la gestión de los eventos.

5.1.4 Componentes y eventos soportados por el AWT de Java

5.1.4.1 Jerarquía de Componentes

Como todas las clases de ***Java***, los componentes utilizados en el AWT pertenecen a una determinada jerarquía de clases, que es muy importante conocer. Esta jerarquía de clases se muestra en la Figura 5.1. Todos los componentes descienden de la clase ***Component***, de la que pueden ya heredar algunos métodos interesantes. El ***package*** al que pertenecen estas clases se llama ***java.awt***.

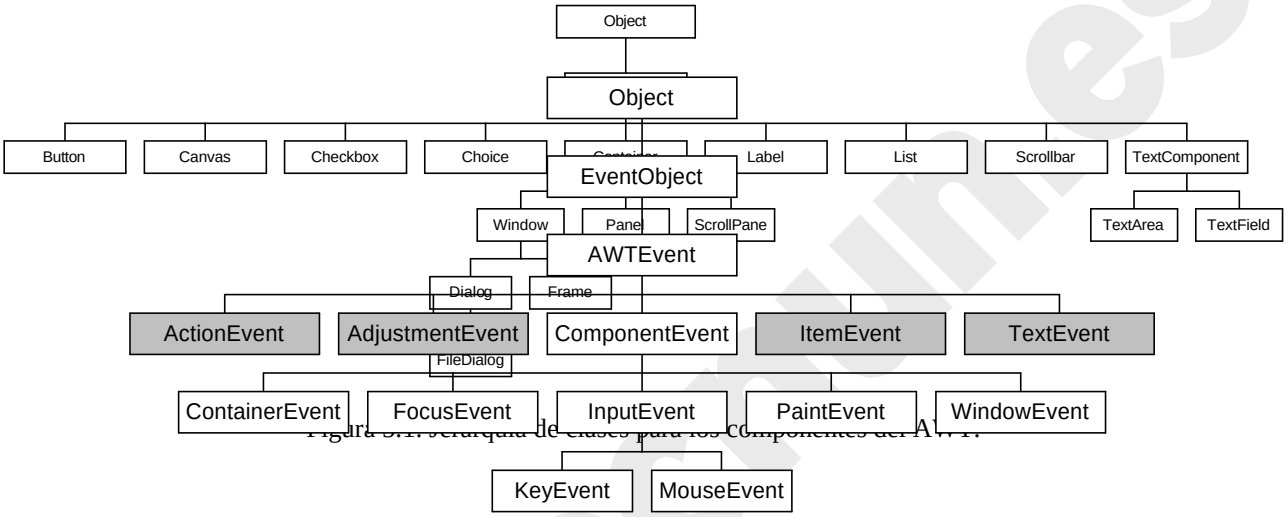


Figura 5.2. Jerarquía de eventos de Java.

A continuación se resumen algunas características importantes de los componentes mostrados en la Figura 5.2:

1. Todos los ***Components*** (excepto ***Window*** y los que derivan de ella) deben ser añadidos a un ***Container***. También un ***Container*** puede ser añadido a otro ***Container***.
2. Para añadir un ***Component*** a un ***Container*** se utiliza el método ***add()*** de la clase ***Container***:

   ```
   containerName.add(componentName);
   ```

3. Los ***Containers*** de máximo nivel son las ***Windows*** (***Frames*** y ***Dialogs***). Los ***Panels*** y ***ScrollPanes*** deben estar siempre dentro de otro ***Container***.
4. Un ***Component*** sólo puede estar en un ***Container***. Si está en un ***Container*** y se añade a otro, deja de estar en el primero.
5. La clase ***Component*** tiene una serie de funcionalidades básicas comunes (variables y métodos) que son heredadas por todas sus ***sub-clases***.

5.1.4.2 Jerarquía de eventos

Todos los eventos de ***Java 1.1*** y ***Java 1.2*** son objetos de clases que pertenecen a una determinada jerarquía de clases. La super-clase ***EventObject*** pertenece al package ***java.util***. De ***EventObject***

deriva la clase ***AWTEvent***, de la que dependen todos los eventos de AWT. La Figura 5.2 muestra la jerarquía de clases para los eventos de ***Java***. Por conveniencia, estas clases están agrupadas en el package ***java.awt.event***.

Los eventos de ***Java*** pueden ser de alto y bajo nivel. Los ***eventos de alto nivel*** se llaman también *eventos semánticos*, porque la acción de la que derivan tiene un significado en sí misma, en el contexto de las interfaces gráficas de usuario. Los ***eventos de bajo nivel*** son las acciones elementales que hacen posible los eventos de alto nivel. Son ***eventos de alto nivel*** los siguientes eventos: los cuatro que tienen que ver con clicar sobre botones o elegir comandos en menús (***ActionEvent***), cambiar valores en barras de desplazamiento (***AdjustmentEvent***), elegir valores (***ItemEvents***) y cambiar el texto (***TextEvent***). En la Figura 5.2 los eventos de alto nivel aparecen con fondo gris.

Los ***eventos de bajo nivel*** son los que se producen con las operaciones elementales con el ratón, teclado, containers y windows. Las seis clases de eventtos de bajo nivel son los eventos relacionados con componentes (***ComponentEvent***), con los containers (***ContainerEvent***), con pulsar teclas (***KeyEvent***), con mover, arrastrar, pulsar y soltar con el ratón (***MouseEvent***), con obtener o perder el focus (***FocusEvent***) y con las operaciones con ventanas (***WindowEvent***).

El modelo de eventos se complica cuando se quiere connstruir un tipo de componente propio, no estándar del AWT. En este caso hay que interceptar los eventos de bajo nivel de ***Java*** y adecuarlos al problema que se trata de resolver. Éste es un tema que no se va a tratar en este manual.

5.1.4.3 Relación entre Componentes y Eventos

La Tabla 5.1 muestra los componentes del AWT y los eventos específicos de cada uno de ellos, así como una breve explicación de en qué consiste cada tipo de evento.

Component	**Eventos generados**	**Significado**
Button	ActionEvent	Clicar en el botón
Checkbox	ItemEvent	Seleccionar o deseleccionar un item
CheckboxMenuItem	ItemEvent	Seleccionar o deseleccionar un item
Choice	ItemEvent	Seleccionar o deseleccionar un item
Component	ComponentEvent	Mover, cambiar tamaño, mostrar u ocultar un componente
	FocusEvent	Obtener o perder el focus
	KeyEvent	Pulsar o soltar una tecla
	MouseEvent	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón (tener en cuenta que este evento tiene dos Listener)
Container	ContainerEvent	Añadir o eliminar un componente de un container
List	ActionEvent	Hacer doble click sobre un item de la lista
	ItemEvent	Seleccionar o deseleccionar un item de la lista
MunuItem	ActionEvent	Seleccionar un item de un menú
Scrollbar	AdjustementEvent	Cambiar el valor de la scrollbar
TextComponent	TextEvent	Cambiar el texto
TextField	ActionEvent	Terminar de editar un texto pulsando Intro
Window	WindowEvent	Acciones sobre una ventana: abrir, cerrar, iconizar, restablecer e iniciar el cierre

Tabla 5.1. Componentes del AWT y eventos específicos que generan.

La relación entre componentes y eventos indicada en la Tabla 5.1 pueden inducir a engaño si no se tiene en cuenta que los eventos propios de una ***super-clase*** de componentes pueden afectar también a los componentes de sus ***sub-clases***. Por ejemplo, la clase ***TextArea*** no tiene ningún evento específico o propio, pero puede recibir los de su ***super-clase TextComponent***.

La Tabla 5.2 muestra los ***componentes*** del AWT y ***todos los tipos de eventos*** que se pueden producir sobre cada uno de ellos, teniendo en cuenta también los que son específicos de sus ***super-clases***. Entre ambas tablas se puede sacar una idea bastante precisa de qué tipos de eventos están soportados en ***Java*** y qué eventos concretos puede recibir cada componente del AWT. En la

AWT Components	**Eventos que se pueden generar**									
	AtionEvent	**AdjustementEvent**	**ComponentEvent**	**ContainerEvent**	**FocusEvent**	**ItemEvent**	**KeyEvent**	**MouseEvent**	**TextEvent**	**WindowEvent**
Button	✓		✓		✓		✓	✓		
Canvas			✓		✓		✓	✓		
Checkbox			✓		✓	✓	✓	✓		
Checkbox-MenuItem						✓				
Choice			✓		✓	✓	✓	✓		
Component			✓		✓		✓	✓		
Container			✓	✓	✓		✓	✓		
Dialog			✓	✓	✓		✓	✓		✓
Frame			✓	✓	✓		✓	✓		✓
Label			✓		✓		✓	✓		
List	✓		✓		✓	✓	✓	✓		
MenuItem	✓									
Panel			✓	✓	✓		✓	✓		
Scrollbar		✓	✓		✓		✓	✓		
TextArea			✓		✓		✓	✓	✓	
TextField	✓		✓		✓		✓	✓	✓	
Window			✓	✓	✓		✓	✓		✓

Tabla 5.2. Eventos que generan los distintos componentes del AWT.

práctica, no todos los tipos de evento tienen el mismo interés.

5.1.5 Interfaces Listener

Una vez vistos los distintos eventos que se pueden producir, conviene ver cómo se deben gestionar estos eventos. A continuación se detalla cómo se gestionan los eventos según el modelo de ***Java***:

1. Cada objeto que puede recibir un evento (***event source***), "registra" uno o más objetos para que los gestionen (***event listener***). Esto se hace con un método que tiene la forma,

   ```
   eventSourceObject.addEventListener(eventListenerObject);
   ```

   donde ***eventSourceObject*** es el objeto en el que se produce el evento, y ***eventListenerObject*** es el objeto que deberá gestionar los eventos. La relación entre ambos se establece a través de una interface ***Listener*** que la clase del ***eventListenerObject*** debe implementar. Esta interface proporciona la declaración de los métodos que serán llamados cuando se produzca el evento. La interface a implementar depende del tipo de evento. La Tabla 5.3 relaciona los distintos tipos de eventos, con la interface que se debe implementar para gestionarlos. Se indican también los métodos declarados en cada interface.

   Es importante observar la correspondencia entre ***eventos*** e ***interfaces Listener***. Cada evento tiene su interface, excepto el ratón que tiene dos interfaces ***MouseListener*** y ***MouseMotionListener***. La razón de esta duplicidad de interfaces se encuentra en la peculiaridad de los eventos que se producen cuando el ratón se mueve. Estos eventos, que se producen con muchísima más frecuencia que los simples clicks, por razones de eficiencia son gestionados por una interface especial: ***MouseMotionListener***.

   Obsérvese que el ***nombre de la interface*** coincide con el ***nombre del evento***, sustituyendo la palabra ***Event*** por ***Listener***.

2. Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente la correspondiente interface ***Listener***, se deben definir los métodos de dicha interface. Siempre hay que definir ***todos los métodos*** de la interface, aunque algunos de dichos métodos puedan estar "vacíos". Un ejemplo es la implementación de la interface ***WindowListener*** vista en el Apartado 1.3.9 (en la página 18), en el que todos los métodos estaban vacíos excepto ***windowClosing()***.

Evento	Interface Listener	Métodos de Listener
ActionEvent	ActionListener	actionPerformed()
AdjustementEvent	AdjustementListener	adjustementValueChanged()
ComponentEvent	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
ContainerEvent	ContainerListener	componentAdded(), componentRemoved()
FocusEvent	FocusListener	focusGained(), focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
	MouseMotionListener	mouseDragged(), mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

Tabla 5.3. Métodos relacionados con cada evento a través de una interface Listener.

5.1.6 Clases Adapter

Java proporciona ayudas para definir los métodos declarados en las interfaces ***Listener***. Una de estas ayudas son las clases ***Adapter***, que existen para cada una de las interfaces ***Listener*** que tienen más de un método. Su nombre se construye a partir del nombre de la interface, sustituyendo la palabra "*Listener*" por "*Adapter*". Hay 7 clases ***Adapter***: *ComponentAdapter*, *ContainerAdapter*, *FocusAdapter*, *KeyAdapter*, *MouseAdapter*, *MouseMotionAdapter* y *WindowAdapter*.

Las clases ***Adapter*** derivan de ***Object***, y son clases predefinidas que contienen ***definiciones vacías*** para todos los métodos de la interface. Para crear un objeto que responda al evento, en vez de crear una clase que implemente la interface ***Listener***, basta crear una clase que derive de la clase ***Adapter*** correspondiente, y redefina sólo los métodos de interés. Por ejemplo, la clase ***VentanaCerrable*** del Apartado 1.3.9 (página 18) se podía haber definido de la siguiente forma:

```
1.    // Fichero VentanaCerrable2.java

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

4.    class VentanaCerrable2 extends Frame {

5.        // constructores
6.        public VentanaCerrable2() { super(); }
7.        public VentanaCerrable2(String title) {
8.            super(title);
9.            setSize(500,500);
10.           CerrarVentana cv = new CerrarVentana();
11.           this.addWindowListener(cv);
12.       }

13.   } // fin de la clase VentanaCerrable2

14.   // definición de la clase CerrarVentana
15.   class CerrarVentana extends WindowAdapter {
16.       void windowClosing(WindowEvent we) { System.exit(0); }
17.   } // fin de la clase CerrarVentana
```

Las sentencias 15-17 definen una clase auxiliar (***helper class***) que deriva de la clase ***WindowAdapter***. Dicha clase hereda definiciones vacías de todos los métodos de la interface ***WindowListener***. Lo único que tiene que hacer es redefinir el único método que se necesita para cerrar las ventanas. El constructor de la clase ***VentanaCerrable*** crea un objeto de la clase ***CerrarVentana*** en la sentencia 10 y lo registra como ***event listener*** en la sentencia 11. En la sentencia 11 la palabra ***this*** es opcional: si no se incluye, se supone que el ***event source*** es el objeto de la clase en la que se produce el evento, en este caso la propia ventana.

Todavía hay otra forma de responder al evento que se produce cuando el usuario desea cerrar la ventana. Las ***clases anónimas*** de ***Java*** son especialmente útiles en este caso. En realidad, para gestionar eventos sólo hace falta un objeto que sea registrado como ***event listener*** y contenga los métodos adecuados de la interface ***Listener***. Las ***clases anónimas*** son útiles cuando sólo se necesita un objeto de la clase, como es el caso. La nueva definición de la clase ***VentanaCerrable*** podría ser como sigue:

```
1.    // Fichero VentanaCerrable3.java

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

```
4.     class VentanaCerrable3 extends Frame {

5.         // constructores
6.         public VentanaCerrable3() { super(); }
7.         public VentanaCerrable3(String title) {
8.             super(title);
9.             setSize(500,500);
10.            this.addWindowListener(new WindowAdapter() {
11.                                       public void windowClosing() {System.exit(0);}
12.                                   });
13.        }

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

Obsérvese que el objeto ***event listener*** se crea justamente en el momento de pasárselo como argumento al método ***addWindowListener()***. Se sabe que se está creando un nuevo objeto porque aparece la palabra ***new***. Debe tenerse en cuenta que no se está creando un nuevo objeto de ***WindowAdapter*** (entre otras cosas porque dicha clase es ***abstract***), sino extendiendo la clase ***WindowAdapter***, aunque la palabra ***extends*** no aparezca. Esto se sabe por las ***llaves*** que se abren al final de la línea 10. Los ***paréntesis vacíos*** de la línea 10 podrían contener los argumentos para el constructor de ***WindowAdapter***, en el caso de que dicho constructor necesitara argumentos. En la sentencia 11 se redefine el método ***windowClosing()***. En la línea 12 se cierran las llaves de la ***clase anónima***, se cierra el paréntesis del método ***addWindowListener()*** y se pone el ***punto y coma*** de terminación de la sentencia que empezó en la línea 10. Para más información sobre las ***clases anónimas*** ver el Apartado 3.10.4, en la página 59.

5.2 COMPONENTES Y EVENTOS

En este Apartado se van a ver los ***componentes gráficos*** de ***Java***, a partir de los cuales se pueden construir interfaces gráficas de usuario. Se verán también, en paralelo y lo más cerca posible en el texto, las diversas clases de ***eventos*** que pueden generar cada uno de esos componentes.

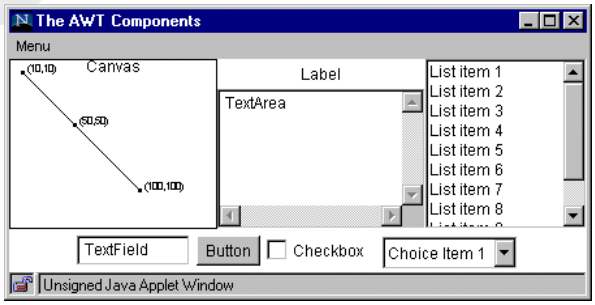


Figura 5.3. Algunos componentes del AWT.

La Figura 5.3, tomada de uno de los ejemplos de ***Java Tutorial*** de ***Sun***, muestra algunos componentes del AWT. En ella se puede ver un *menú*, una superficie de dibujo o *canvas* en la que se puede dibujar y escribir texto, una *etiqueta*, una *caja de texto* y un *área de texto*, un *botón de comando* y un *botón de selección*, una *lista* y una *caja de selección desplegable.*

5.2.1 Clase Component

Métodos de Component	**Función que realizan**
boolean isVisible(), void setVisible(boolean)	Permiten chequear o establecer la visibilidad de un componente
boolean isShowing()	Permiten saber si un componente se está viendo. Para ello tanto el componente debe ser visible, y su container debe estar mostrándose
boolean isEnabled(), void setEnabled(boolean)	Permiten saber si un componente está activado y activarlo o desactivarlo
Point getLocation(), Point getLocationOnScreen()	Permiten obtener la posición de la esquina superior izquierda de un componente respecto al componente-padre o a la pantalla
void setLocation(Point), void setLocation(int x, int y)	Desplazan un componente a la posición especificada respecto al container o componente-padre
Dimension getSize(), void setSize(int w, int h), void setSize(Dimension d)	Permiten obtener o establecer el tamaño de un componente
Rectangle getBounds(), void setBounds(Rectangle), void setBounds(int x, int y, int width, int height)	Obtienen o establecen la posición y el tamaño de un componente
invalidate(), validate(), doLayout()	invalidate() marca un componente y sus contenedores para indicar que se necesita volver a aplicar el Layout Manager. validate() se asegura que el Layout Manager está bien aplicado. doLayout() hace que se aplique el Layout Manager
paint(Graphics), repaint() y update(Graphics)	Métodos gráficos para dibujar en la pantalla
setBackground(Color), setForeground(Color)	Métodos para establecer los colores por defecto

Tabla 5.4. Métodos de la clase Component.

La clase ***Component*** es una clase ***abstract*** de la que derivan todas las clases del AWT, según el diagrama mostrado previamente en la Figura 5.1, en la página 89. Los métodos de esta clase son importantes porque son heredados por todos los componentes del AWT. La Tabla 5.4 muestra algunos de los métodos más utilizados de la clase ***Component***. En las declaraciones de los métodos de dicha clase aparecen las clases ***Point***, ***Dimension*** y ***Rectangle***. La clase ***java.awt.Point*** tiene dos variables miembro *int* llamadas ***x*** e ***y***. La clase ***java.awt.Dimension*** tiene dos variables miembro *int*: ***height*** y ***width***. La clase ***java.awt.Rectangle*** tiene cuatro variables *int*: ***height***, ***width***, ***x*** e ***y***. Las tres son sub-clases de ***Object***.

Además de los métodos mostrados en la Tabla 5.4, la clase ***Component*** tiene un gran número de métodos básicos cuya funcionalidad puede estudiarse mediante la documentación on-line de ***Java***. Entre otras funciones, permiten controlar los ***colores***, las ***fonts*** y los ***cursores***.

A continuación se describen las clases de ***eventos*** más generales, relacionados bien con la clase ***Component***, bien con diversos tipos de componentes que también se presentan a continuación.

5.2.2 Clases EventObject y AWTEvent

Todos los métodos de las interfaces ***Listener*** relacionados con el AWT tienen como argumento único un objeto de alguna clase que desciende de la clase ***java.awt.AWTEvent*** (ver Figura 5.2, en la página 89).

La clase ***AWTEvent*** desciende de ***java.util.EventObject***. La clase ***AWTEvent*** no define ningún método, pero hereda de ***EventObject*** el método ***getSource()***:

```
Object getSource();
```

que devuelve una referencia al objeto que generó el evento. Las clases de eventos que descienden de ***AWTEvent*** definen métodos similares a ***getSource()*** con unos valores de retorno menos genéricos. Por ejemplo, la clase ***ComponentEvent*** define el método ***getComponent()***, cuyo valor de retorno es un objeto de la clase ***Component***.

5.2.3 Clase ComponentEvent

Los eventos ***ComponentEvent*** se generan cuando un ***Component*** de cualquier tipo se muestra, se oculta, o cambia de posición o de tamaño. Los eventos de *mostrar* u *ocultar* ocurren cuando se llama al método ***setVisible(boolean)*** del ***Component***, pero no cuando se *minimiza* la ventana.

Otro método útil de la clase ***ComponentEvent*** es ***Component getComponent()*** que devuelve el componente que generó el evento. Se puede utilizar en lugar de ***getSource()***.

5.2.4 Clases InputEvent y MouseEvent

De la clase ***InputEvent*** descienden los eventos del ratón y el teclado. Esta clase dispone de métodos pera detectar si los botones del ratón o las teclas especiales han sido pulsadas. Estos botones y estas teclas se utilizan para cambiar o modificar el significado de las acciones del usuario. La clase ***InputEvent*** define unas constantes que permiten saber qué teclas especiales o botones del ratón estaban pulsados al producirse el evento, como son: SHIFT_MASK, ALT_MASK, CTRL_MASK, BUTTON1_MASK, BUTTON2_MASK y BUTTON3_MASK, cuyo significado es evidente. La Tabla 5.5 muestra algunos métodos de esta clase.

Métodos heredados de la clase InputEvent	**Función que realizan**
boolean isShiftDown(), boolean isAltDown(), boolean isControlDown()	Devuelven un boolean con información sobre si esa tecla estaba pulsada o no
int getModifiers()	Obtiene información con una máscara de bits sobre las teclas y botones pulsados
long getWhen()	Devuelve la hora en que se produjo el evento

Tabla 5.5. Métodos de la clase InputEvent.

Se produce un ***MouseEvent*** cada vez que el cursor movido por el ratón entra o sale de un componente visible en la pantalla, al clicar, o cuando se pulsa o se suelta un botón del ratón. Los métodos de la interface ***MouseListener*** se relacionan con estas acciones, y son los siguientes (ver Tabla 5.3, en la página 92): ***mouseClicked()***, ***mouseEntered()***, ***mouseExited()***, ***mousePressed()*** y ***mouseReleased()***. Todos son *void* y reciben como argumento un objeto ***MouseEvent***. La Tabla 5.6 muestra algunos métodos de la clase ***MouseEvent***.

Métodos de la clase MouseEvent	**Función que realizan**
int getClickCount()	Devuelve el número de clicks en ese evento
Point getPoint(), int getX(), int getY()	Devuelven la posición del ratón al producirse el evento
boolean isPopupTrigger()	Indica si este evento es el que dispara los menús popup

Tabla 5.6. Métdos de la clase MouseEvent.

La clase ***MouseEvent*** define una serie de constantes ***int*** que permiten identificar los tipos de eventos que se han producido: MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_DRAGGED.

Además, el método ***Component getComponent()***, heredado de ***ComponentEvent***, devuelve el componente sobre el que se ha producido el evento.

Los eventos ***MouseEvent*** disponen de una segunda interface para su gestión, la interface ***MouseMotionListener***, cuyos métodos reciben también como argumento un evento de la clase ***MouseEvent***. Estos eventos están relacionados con el ***movimiento del ratón***. Se llama a un método de la interface ***MouseMotionListener*** cuando el usuario utiliza el ratón (o un dispositivo similar) para mover el cursor o arrastrarlo sobre la pantalla. Los métodos de la interface ***MouseMotionListener*** son ***mouseMoved()*** y ***mouseDragged()***.

5.2.5 Clase FocusEvent

El ***Focus*** está relacionado con la posibilidad de sustituir al ratón por el teclado en ciertas operaciones. De los componentes que aparecen en pantalla, en un momento dado hay sólo uno que puede recibir las acciones del teclado y se dice que ese componente tiene el ***Focus***. El componente que tiene el ***Focus*** aparece diferente de los demás (resaltado de alguna forma). Se cambia el elemento que tiene el ***Focus*** con la tecla ***Tab*** o con el ratón. Se produce un ***FocusEvent*** cada vez que un componente gana o pierde el ***Focus***.

El método ***requestFocus()*** de la clase ***Component*** permite hacer desde el programa que un componente obtenga el ***Focus***.

El método ***boolean isTemporary()***, de la clase ***FocusEvent***, indica si la pérdida del ***Focus*** es o no temporal (puede ser temporal por haberse ocultado o dejar de estar activa la ventana, y recuperarse al cesar esta circunstancia).

El método ***Component getComponent()*** es heredado de ***ComponentEvent***, y permite conocer el componente que ha ganado o perdido el ***Focus***. Las constantes de esta clase ***FOCUS_GAINED*** y ***FOCUS_LOST*** permiten saber el tipo de evento ***FocusEvent*** que se ha producido.

5.2.6 Clase Container

La clase ***Container*** es también una clase muy general. De ordinario, nunca se crea un objeto de esta clase, pero los métodos de esta clase son heredados por las clases ***Frame*** y ***Panel***, que sí se utilizan con mucha frecuencia para crear objetos. La Tabla 5.7 muestra algunos métodos de ***Container***.

Los containers mantienen una ***lista de los objetos*** que se les han ido añadiendo. Cuando se añade un nuevo objeto se incorpora al final de la lista, salvo que se especifique una posición determinada. En esta clase tiene mucha importancia todo lo que tiene que ver con los ***Layout Managers***, que se explicarán más adelante. La Tabla 5.7 muestra algunos métodos de la clase ***Container***.

5.2.7 Clase ContainerEvent

Métodos de Container	**Función que realizan**
Component add(Component)	Añade un componente al container
doLayout()	Ejecuta el algoritmo de ordenación del layout manager
Component getComponent(int)	Obtiene el n-ésimo componente en el container
Component getComponentAt(int, int), Component getComponentAt(Point)	Obtiene el componente que contine un determinado punto
int getComponentCount()	Obtiene el número de componentes en el container
Component[] getComponents()	Obtiene los componentes en este container.
remove(Component), remove(int), removeAll()	Elimina el componente especificado.
setLayout(LayoutManager)	Determina el layout manager para este container

Tabla 5.7. Métodos de la clase Container.

Los ***ContainerEvents*** se generan cada vez que un ***Component*** se añade o se retira de un ***Container***. Estos eventos sólo tienen un *papel de aviso* y no es necesario gestionarlos para que se realize la operación.

Los métodos de esta clase son ***Component getChild()***, que devuelve el ***Component*** añadido o eliminado, y ***Container getContainer()***, que devuelve el ***Container*** que generó el evento.

5.2.8 Clase Window

Los objetos de la clase ***Window*** son ventanas de máximo nivel, pero ***sin bordes*** y ***sin barra de menús***. En realidad son más interesantes las clases que derivan de ella: ***Frame*** y ***Dialog***. Los métodos más útiles, por ser heredados por las clases ***Frame*** y ***Dialog***, se muestran en la Tabla 5.8.

Métodos de Window	**Función que realizan**
toFront(), toBack()	Para desplazar la ventana hacie adelante y hacia atrás en la pantalla
show()	Muestra la ventana y la trae a primer plano
pack()	Hace que los componentes se reajusten al tamaño preferido

Tabla 5.8. Métodos de la clase Window.

5.2.9 Clase WindowEvent

Se produce un ***WindowEvent*** cada vez que se *abre*, *cierra*, *iconiza*, *restaura*, *activa* o *desactiva* una ventana. La interface ***WindowListener*** contiene los siete métodos siguientes, con los que se puede responder a este evento:

```
void windowOpened(WindowEvent we);  // antes de mostrarla por primera vez
void windowClosing(WindowEvent we); // al recibir una solicitud de cierre
void windowClosed(WindowEvent we);  // después de cerrar la ventana
```

```
void windowIconified(WindowEvent we);
void windowDeiconified(WindowEvent we);
void windowActivated(WindowEvent we);
void windowDeactivated(WindowEvent we);
```

El uso más frecuente de ***WindowEvent*** es para cerrar ventanas (por defecto, los objetos de la clase ***Frame*** no se pueden cerrar más que con ***Ctrl+Alt+Supr***). También se utiliza para detener ***threads*** y liberar recursos al iconizar una ventana (que contiene por ejemplo animaciones) y comenzar de nuevo al restaurarla.

La clase ***WindowEvent*** define la siguiente serie de constantes que permiten identificar el tipo de evento:

WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED,
WINDOW_ICONIFIED, WINDOW_DEICONIFIED,
WINDOW_ACTIVATED, WINDOW_DEACTIVATED

En la clase ***WindowEvent*** el método ***Window getWindow()*** devuelve la ***Window*** que generó el evento. Se utiliza en lugar de ***getSource()***.

5.2.10 Clase Frame

Es una ventana ***con un borde*** y que puede tener una ***barra de menús***. Si una ventana depende de otra ventana, es mejor utilizar una ***Window*** (ventana sin borde ni barra de menús) que un ***Frame***. La Tabla 5.9 muestra algunos métodos más utilizados de la clase ***Frame***.

Métodos de Frame	**Función que realiza**
Frame(), Frame(String title)	Constructores de Frame
String getTitle(), setTitle(String)	Obtienen o determinan el título de la ventana
MenuBar getMenuBar(), setMenuBar(MenuBar), remove(MenuComponent)	Permite obtener, establecer o eliminar la barra de menús
Image getIconImage(), setIconImage(Image)	Obtienen o determinan el icono que apareceré en la barra de títulos
setResizable(boolean), boolean isResizable()	Determinan o chequean si se puede cambiar el tamaño
dispose()	Método que libera los recursos utilizados en una ventana. Todos los componentes de la ventana son destruidos.

Tabla 5.9. Métodos de la clase Frame.

Además de los métodos citados, se utilizan mucho los métodos ***show()***, ***pack()***, ***toFront()*** y ***toBack()***, heredados de la super-clase ***Window***.

5.2.11 Clase Dialog

Un ***Dialog*** es una ventana que depende de otra ventana (de una ***Frame***). Si una ***Frame*** se cierra, se cierran también los ***Dialog*** que dependen de ella; si se iconifica, sus ***Dialog*** desaparecen; si se restablece, sus ***Dialog*** aparecen de nuevo. Este comportamiento se obtiene de forma automática.

Las ***Applets*** estándar no soportan ***Dialogs*** porque no son ***Frames*** de ***Java***. Las ***Applets*** que abren ***Frames*** sí pueden soportar ***Dialogs***.

Un ***Dialog modal*** require la atención inmediata del usuario: no se puede hacer ninguna otra cosa hasta no haber cerrado el ***Dialog***. Por defecto, los ***Dialogs*** son ***no modales***. La Tabla 5.10 muestra los métodos más importantes de la clase ***Dialog***. Se pueden utilizar también los métodos heredados de sus super-clases.

Métodos de Dialog	**Función que realiza**
Dialog(Frame fr), Dialog(Frame fr, boolean mod), Dialog(Frame fr, String title), Dialog(Frame fr, String title, boolean mod)	Constructores
String getTitle(), setTitle(String)	Permite obtener o determinar el título
boolean isModal(), setModal(boolean)	Pregunta o determina si el Dialog es modal o no
boolean isResizable(), setResizable(boolean)	Pregunta o determina si se puede cambiar el tamaño
show()	Muestra y trae a primer plano el Dialog

Tabla 5.10. Métodos de la clase Dialog.

5.2.12 Clase FileDialog

La clase ***FileDialog*** muestra una ventana de diálogo en la cual se puede seleccionar un fichero. Esta clase deriva de ***Dialog***. Las constantes enteras LOAD (abrir ficheros para lectura) y SAVE (abrir ficheros para escritura) definen el ***modo*** de apertura del fichero. La Tabla 5.11 muestra algunos métodos de esta clase.

Métodos de la clase FileDialog	**Función que realizan**
FileDialog(Frame parent), FileDialog(Frame parent, String title), public FileDialog(Frame parent, String title, int mode)	Constructores
int getMode(),setMode(int mode)	Modo de apertura (SAVE o LOAD)
String getDirectory(), String getFile()	Obtiene el directorio o fichero elegido
setDirectory(String dir), setFile(String file)	Determina el directorio o fichero elegido
FilenameFilter getFilenameFilter(), setFilenameFilter(FilenameFilter filter)	Determina o establece el filtro para los ficheros

Tabla 5.11. Métodos de la clase FileDialog.

Las clases que implementan la interface ***java.io.FilenameFilter*** permiten filtrar los ficheros de un directorio. Para más información, ver la docomentación on-line.

5.2.13 Clase Panel

Un ***Panel*** es un ***Container*** de propósito general. Se puede utilizar tal cual para contener otras componentes, y también crear una sub-clase para alguna finalidad más específica. Por defecto, el ***Layout Manager*** de ***Panel*** es ***FlowLayout***. Los ***Applets*** son sub-clases de ***Panel***. La Tabla 5.12 muestra los métodos más importantes que se utilizan con la clase ***Panel***, que son algunos métodos heredados de ***Componet*** y ***Container***, pues la clase ***Panel*** no tiene métodos propios.

Métodos de Panel	**Función que realiza**
Panel(), Panel(LayoutManager miLM)	Constructores de Panel
Métodos heredados de Container y Component	
add(Component), add(Component, int)	Añade componentes al panel
setLayout(), getLayout()	Establece o permite obtener el layout manager utilizado
validate(), doLayout()	Para reorganizar los componentes después de algún cambio. Es mejor utilizar validate()
remove(int), remove(Component), removeAll()	Para eliminar componentes
Dimension getMaximumSize(), Dimension getMinimumSize(), Dimension getPreferredSize()	Permite obtener los tamaños máximo, mínimo y preferido
Insets getInsets()	

Tabla 5.12. Métodos de la clase Panel.

Un ***Panel*** puede contener otros ***Panel***. Esto es una gran ventaja respecto a los demás tipos de containers, que son containers de máximo nivel y no pueden introducirse en otros containers.

Insets es una clase que deriva de ***Object***. Sus variables son ***top***, ***left***, ***botton***, ***right***. Representa el *espacio que se deja libre* en los bordes de un ***Container***. Se establece mediante la llamada al adecuado constructor del ***Layout Manager***.

5.2.14 Clase Button

Aunque según la Tabla 5.2, en la página 91, un ***Button*** puede recibir seis tipos de eventos, lo más importante es que al clicar sobre él se genera un evento de la clase ***ActionEvent***.

El aspecto de un ***Button*** depende de la plataforma (PC, Mac, Unix), pero la funcionalidad siempre es la misma. Se puede cambiar el ***texto*** y la ***font*** que aparecen en el ***Button***, así como el ***foreground*** y ***backgroung color***. También se puede establecer que esté activado o no. La Tabla 5.13 muestra los métodos más importantes de la clase ***Button***.

Métodos de la clase Button	**Función que realiza**
Button(String label) y Button()	Constructores
setLabel(String str), String getLabel()	Permite establecer u obtener la etiqueta del Button
addActionListener(ActionListener al), removeActionListener(ActionListener al)	Permite registrar el objeto que gestionará los eventos, que deberá implementar ActionListener
setActionCommand(String cmd), String getActionCommand()	Establece y recupera un nombre para el objeto Button independiente del label y del idioma

Tabla 5.13. Métodos de la clase Button.

5.2.15 Clase ActionEvent

Los eventos ***ActionEvent*** se producen al clicar con el ratón en un botón (***Button***), al elegir un comando de un menú (***MenuItem***), al hacer doble clic en un elemento de una lista (***List***) y al pulsar ***Intro*** para introducir un texto en una caja de texto (***TextField***).

El método ***String getActionCommand()*** devuelve el texto asociado con la acción que provocó el evento. Este texto se puede fijar con el método ***setActionCommand(String str)*** de las clases ***Button*** y ***MenuItem***. Si el texto no se ha fijado con este método, el método ***getActionCommand()*** devuelve el texto mostrado por el componente (su etiqueta). Para objetos con varios items el valor devuelto es el nombre del item seleccionado.

El método ***int getModifiers()*** devuelve un entero representando una constante definida en ***ActionEvent*** (SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK). Estas constantes sirven para determinar si se pulsó una de estas teclas modificadores mientras se clicaba. Por ejemplo, si se estaba pulsando la tecla CTRL la siguiente expresión es distinta de cero:

```
actionEvent.getModifiers() & ActionEvent.CTRL_MASK
```

5.2.16 Clase Canvas

Una ***Canvas*** es una zona rectangular de pantalla en la que se puede dibujar y en la que se pueden generar eventos. Las ***Canvas*** permiten realizar dibujos, mostrar imágenes y crear componentes a medida, de modo que muestren un aspecto similar en todas las plataformas. La Tabla 5.14 muestra

Métodos de Canvas	**Función que realiza**
Canvas()	Es el único constructor de esta clase
paint(Graphics g);	Dibuja un rectángulo con el color de background. Lo normal es que las sub-clases de Canvas redefinan este método.

Tabla 5.14. Métodos de la clase Canvas.

los métodos de la clase ***Canvas***.

Desde los objetos de la clase ***Canvas*** se puede llamar a los métodos ***paint()*** y ***repaint()*** de la super-clase ***Component***. Con frecuencia conviene redefinir los siguientes métodos de ***Component***: ***getPreferredSize()***, ***getMinimumSize()*** y ***getMaximumSize()***, que devuelven un objeto de la clase ***Dimension***. El ***LayoutManager*** se encarga de utilizar estos valores.

La clase ***Canvas*** no tiene eventos propios, pero puede recibir los eventos ***ComponentEvent*** de su super-clase ***Component***.

5.2.17 Component Checkbox y clase CheckboxGroup

Los objetos de la clase ***Checkbox*** son ***botones de opción o de selección*** con dos posibles valores: ***on*** y ***off***. Al cambiar la selección de un ***Checkbox*** se produce un ***ItemEvent***.

Métodos de Checkbox	**Función que realizan**
Checkbox(), Checkbox(String), Checkbox(String, boolean), Checkbox(String, boolean, CheckboxGroup), Checkbox(String, CheckboxGroup, boolean)	Constructores de Checkbox. Algunos permiten establecer la etiqueta, si está o no seleccionado y si pertenece a un grupo
addItemListener(ItemListener), removeItemListener(ItemListener)	Registra o elimina los objetos que gestionarán los eventos ItemEvent
setLabel(String), String getLabel()	Establece u obtiene la etiqueta del componente
setState(boolean), boolean getState()	Establece u obtiene el estado (true o false, según esté seleccionado o no)
setCheckboxGroup(CheckboxGroup), CheckboxGroup getCheckboxGroup()	Establece u obtiene el grupo al que pertenece el Checkbox
Métodos de CheckboxGroup	**Función que realizan**
CheckboxGroup()	Constructores de CheckboxGroup:
Checkbox getSelectedCheckbox(), setSelectedCheckbox(Checkbox box)	Obtiene o establece el componente seleccionado de un grupo:

Tabla 5.15. Métodos de las clases Checkbox y CheckboxGroup.

La clase ***CheckboxGroup*** permite la opción de agrupar varios ***Checkbox*** de modo que uno y sólo uno esté en ***on*** (al comienzo puede que todos estén en ***off***). Se corresponde con los botones de opción de Visual Basic. La Tabla 5.15 muestra los métodos más importantes de estas clases.

Cuando el usuario actúa sobre un objeto ***Checkbox*** se ejecuta el método ***itemStateChanged()***, que es el único método de la interface ***ItemListener***. Si hay varias ***checkboxes*** cuyos eventos se gestionan en un mismo objeto y se quiere saber cuál es la que ha recibido el evento, se puede utilizar el método ***getSource()*** del evento ***EventObject***. También se puede utilizar el método ***getItem()*** de ***ItemEvent***, cuyo valor de retorno es un ***Object*** que contiene el ***label*** del componente (para convertirlo a ***String*** habría que hacer un ***cast***).

Para crear un ***grupo*** o conjunto de botones de opción (de forma que uno y sólo uno pueda estar activado), se debe crear un objeto de la clase ***CheckboxGroup***. Este objeto no tiene datos: simplemente sirve como identificador del grupo. Cuando se crean los objetos ***Checkbox*** se pasa a los ***constructores*** el objeto ***CheckboxGroup*** del grupo al que se quiere que pertenezcan.

Cuando se selecciona un ***Checkbox*** de un grupo se producen dos eventos: uno por el elemento que se ha seleccionado y otro por haber perdido la selección el elemento que estaba seleccionado anteriormente. Al hablar de evento ***ItemEvent*** y del método ***itemStateChanged()*** se verán métodos para determinar los ***checkboxes*** que han sido seleccionados o que han perdido la selección.

5.2.18 Clase ItemEvent

Se produce un ***ItemEvent*** cuando ciertos componentes (***Checkbox***, ***CheckboxMenuItem***, ***Choice*** y ***List***) cambian de estado (on/off). Estos componentes son los que implementan la interface ***ItemSelectable***. La Tabla 5.16 muestra algunos métodos de esta clase.

Métodos de la clase ItemEvent	Función que realizan
Object getItem()	Devuelve el objeto donde se originó el evento
ItemSelectable getItemSelectable()	Devuelve el objeto ItemSelectable donde se originó el evento
int getStateChange()	Devuelve una de las constantes SELECTED o DESELECTED definidas en la clase ItemEvent

Tabla 5.16. Métodos de la clase ItemEvent.

La clase ***ItemEvent*** define las constantes enteras SELECTED y DESELECTED, que se pueden utilizar para comparar con el valor devuelto por el método ***getStateChange()***.

5.2.19 Clase Choice

La clase ***Choice*** permite elegir un item de una lista desplegable. Los objetos ***Choice*** ocupan menos espacio en pantalla que los ***Checkbox***. Al elegir un item se genera un ***ItemEvent***. Un ***index*** permite determinar un elemento de la lista (se empieza a contar desde 0). La Tabla 5.17 muestra los métodos más habituales de esta clase.

Métodos de Choice	Función que realizan
Choice()	Constructor de Choice
addItemListener(ItemListener), removeItemListener(ItemListener)	Establece o elimina un ItemListener
add(String), addItem(String)	Añade un elemento a la lista
insert(String label, int index)	Inserta un elemento con un label an la posición incicada
int getSelectedIndex(), String getSelectedItem()	Obtiene el index o el label del elemento elegido de la lista
int getItemCount()	Obtiene el número de elementos
String getItem(int)	Obtiene el label a partir del index
select(int), select(String)	Selecciona un elemento por el index o el label
removeAll(), remove(int), remove(String)	Elimina todos o uno de los elementos de la lista

Tabla 5.17. Métodos de la clase Choice.

La clase ***Choice*** genera el evento ***ItemEvent*** al seleccionar un item de la lista. En este sentido es similar a las clases ***Checkbox*** y ***CheckboxGroup***, así como a los menús de selección.

5.2.20 Clase Label

La clase ***Label*** introduce en un container un ***texto no seleccionable y no editable***, que por defecto se alinea por la izquierda. La clase ***Label*** define las constantes Label.CENTER, Label.LEFT y Label.RIGHT para determinar la alineación del texto. La Tabla 5.18 muestra algunos métodos de esta clase

Métodos de Label	**Función que realizan**
Label(String lbl), Label(String lbl, int align)	Constructores de Label
setAlignement(int align), int getAlignement()	Establecer u obtener la alineación del texto
setText(String txt), String getText()	Establecer u obtener el texto del Label

Tabla 5.18. Métodos de la clase Label.

La elección del *font*, de los *colores*, del tamaño y posición de ***Label*** se realiza con los métodos heredados de la clase ***Component***: ***setFont(Font f)***, ***setForeground(Color)***, ***setBackground(Color)***, ***setSize(int, int)***, ***setLocation(int, int)***, ***setVisible(boolean)***, etc.

La clase ***Label*** no tiene más ***eventos*** que los de su super-clase ***Component***.

5.2.21 Clase List

La clase ***List*** viene definida por una zona de pantalla con varias líneas, de las que se muestran sólo algunas, y entre las que se puede hacer una ***selección simple o múltiple***. Las ***List*** generan eventos de la clase ***ActionEvents*** (al ***clicar dos veces*** sobre un item o al pulsar ***return***) e ***ItemEvents*** (al seleccionar o deseleccionar un item). Al gestionar el evento ***ItemEvent*** se puede preguntar si el usuario estaba pulsando a la vez alguna tecla (***Alt***, ***Ctrl***, ***Shift***), por ejemplo para hacer una selección múltiple.

Las ***List*** se diferencian de las ***Choices*** en que muestran varios items a la vez y que permiten hacer selecciones múltiples. La Tabla 5.19 muestra los principales métodos de la clase ***List***.

Métodos de List	**Función que realiza**
List(), List(int nl), List(int nl, boolean mult)	Constructor: por defecto una línea y selección simple
add(String), add(String, int), addItem(String), addItem(String, int)	Añadir un item. Por defecto se añaden al final
addActionListener(ActionListener), addItemListener(ItemListener)	Registra los objetos que gestionarán los dos tipos de eventos soportados
insert(String, int)	Inserta un nuevo elemento en la lista
replaceItem(String, int)	Sustituye el item en posición int por el String
delItem(int), remove(int), remove(String), removeAll()	Eliminar uno o todos los items de la lista
int getItemCount(), int getRows()	Obtener el número de items o el número de items visibles
String getItem(int), String[] getItems()	Obtiene uno o todos los elementos de la lista
int getSelectedIndex(), String getSelectedItem(), int[] getSelectedIndexes(), String[] getSelectedItems()	Obtiene el/los elementos seleccionados
select(int), deselect(int)	Selecciona o elimina la selección de un elemento
boolean isIndexSelected(int), boolean isItemSelected(String)	Indica si un elemento está seleccionado o no
boolean isMultipleMode(), setMultipleMode(boolean)	Pregunta o establece el modo de selección múltiple
int getVisibleIndex(), makeVisible(int)	Indicar o establecer si un item es visible

Tabla 5.19. Métodos de la clase List.

5.2.22 Clase Scrollbar

Una ***Scrollbar*** es una barra de desplazamiento con un cursor que permite introducir y modificar valores, entre unos valores mínimo y máximo, con pequeños y grandes incrementos. Las ***Scrollbars*** de ***Java*** se utilizan tanto como "sliders" o barras de desplazamiento aisladas (al estilo de Visual Basic), como unidas a una ventana en posición vertical y/u horizontal para mostrar una cantidad de información superior a la que cabe en la ventana.

La clase ***Scrollbar*** tiene dos constantes, Scrollbar.HORIZONTAL y Scrollbar.VERTICAL, que indican la posición de la barra. El cambiar el valor de la ***Scrollbar*** produce un ***AdjustementEvent***. La Tabla 5.20 muestra algunos métodos de esta clase.

Métodos de Scrollbar	**Función que realizan**
Scrollbar(), Scrollbar(int pos), Scrollbar(int pos, int val, int vis, int min, int max)	Constructores de Scrollbar
addAdjustmentListener(AdjustmentListener)	egistra el objeto que gestionará los eventos
int getValue(), setValue(int)	Permiten obtener y fijar el valor
setMaximum(int), setMinimum(int)	Establecen los valores máximo y mínimo
setVisibleAmount(int), int getVisibleAmount()	Establecen y obtienen el tamaño del área visble
setUnitIncrement(int), int getUnitIncrement()	Establecen y obtienen el incremento pequeño
setBlockIncrement(int), int getBlockIncrement()	Establecen y obtienen el incremento grande
setOrientation(int), int getOrientation()	Establecen y obtienen la orientación
setValues(int value, int vis, int min, int max)	Establecen los parámetros de la barra

Tabla 5.20. Métodos de la clase Scrollbar.

En el constructor general, el parámetro ***pos*** es la constante que indica la posición de la barra (horizontal o vertical); el ***rango*** es el intervalo entre los valores mínimo ***min*** y máximo ***max***; el parámetro ***vis*** (de ***visibleAmount***) es el tamaño del área visible en el caso en que las ***Scrollbars*** se utilicen en ***TextAreas***. En ese caso, el tamaño del cursor representa la relación entre el ***área visible*** y el ***rango***, como es habitual en ***Netscape***, ***Word*** y tantas aplicaciones de ***Windows***. El valor seleccionado viene dado por la variable ***value***. Cuando ***value*** es igual a ***min*** el área visible comprende el inicio del rango; cuando ***value*** es igual a ***max*** el área visble comprende el final del ***rango***. Cuando la ***Scrollbar*** se va a utilizar aislada (como ***slider***), se debe hacer ***visibleAmount*** igual a cero.

Las variables ***Unit Increment*** y ***Block Increment*** representan los incrementos pequeño y grande, respectivamente. Por defecto, ***Unit Increment*** es "1" y ***Block Increment*** es "10", mientras que ***min*** es "0" y ***max*** es "100".

Cada vez que cambia el valor de una ***Scrollbar*** se genera un evento ***AdjustementEvent*** y se ejecuta el único método de la interface ***AdjustmentListener***, que es ***adjustmentValueChanged()***.

5.2.23 Clase AdjustmentEvent

Se produce un evento ***AdjustementEvent*** cada vez que se cambia el valor (entero) de una ***Scrollbar***. Hay cinco tipos de ***AdjustementEvent***:

1. ***track***: se arrastra el cursor de la ***Scrollbar***.
2. ***unit increment***, ***unit decrement***: se clica en las flechas de la ***Scrollbar***.

3. ***block increment***, ***block decrement***: se clica encima o debajo del cursor.

Métodos de la clase AdjustementEvent	Función que realizan
Adjustable getAdjustable()	Devuelve el Component que generó el evento (implementa la interface Adjustable)
int getAdjustementType()	Devuelve el tipo de adjustement
int getValue()	Devuelve el valor de la Scrollbar después del cambio

Tabla 5.21. Métodos de la clase AdjustmentEvent.

La Tabla 5.21 muestra algunos métodos de esta clase. Las constantes UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK permiten saber el tipo de acción producida, comparando con el valor de retorno de ***getAdjustementType()***.

5.2.24 Clase ScrollPane

Un ***ScrollPane*** es como una ventana de tamaño limitado en la que se puede mostrar un componente de mayor tamaño con dos ***Scrollbars***, una horizontal y otra vertical. El componente puede ser una imagen, por ejemplo. Las ***Scrollbars*** son visibles sólo si son necesarias (por defecto). Las constantes de la clase ***ScrollPane*** son (su significado es evidente): SCROLLBARS_AS_NEEDED, SCROLLBARS_ALWAYS, SCROLLBARS_NEVER. Los ***ScrollPanes*** no generan eventos. La Tabla 5.22 muestra algunos métodos de esta clase.

Métodos de ScrollPane	Función que realizan
ScrollPane(), ScrollPane(int scbs)	Constructores que pueden incluir las ctes.
Dimension getViewportSize(), int getHScrollbarHeight(), int getVScrollbarWidth()	Obtiene el tamaño del ScrollPane y la altura y anchura de las barras de desplazamiento
setScrollPosition(int x, int y), setScrollPosition(Point p), Point getScrollPosition()	Permiten establecer u obtener la posición del componente
setSize(int, int), add(Component)	Heredados de Container, permiten establecer el tamaño y añadir un componente

Tabla 5.22. Métodos de la clase ScrollPane.

En el caso en que no aparezcan scrollbars (SCROLLBARS_NEVER) será necesario desplazar el componente (hacer ***scrolling***) desde programa, con el método ***setScrollPosition()***.

5.2.25 Clases TextArea y TextField

Ambas componentes heredan de la clase ***TextComponent*** y muestran texto seleccionable y editable. La diferencia principal es que ***TextField*** sólo puede tener una línea, mientras que ***TextArea*** puede tener varias líneas. Además, ***TextArea*** ofrece posibilidades de edición de texto adicionales.

Se pueden especificar el *font* y los *colores de foreground y background*. Sólo la clase ***TextField*** genera ***ActionEvents***, pero como las dos heredan de la clase ***TextComponent*** ambas pueden recibir ***TextEvents***. La Tabla 5.23 muestra algunos métodos de las clases ***TextComponent***, ***TextField*** y ***TextArea***. No se pueden crear objetos de la clase ***TextComponent*** porque su ***constructor*** no es ***public***; por eso su constructor no aparece en la Tabla 5.23.

Métodos heredados de TextComponent	Función que realizan
String getText() y setText(String str)	Permiten establecer u obtener el texto del componente
setEditable(boolean b), boolean isEditable()	Hace que el texto sea editable o pregunta por ello
setCaretPosition(int n), int getCaretPosition()	Fija la posición del punto de inserción o la obtiene
String getSelectedText(), int getSelectionStart() y int getSelectionEnd()	Obtiene el texto seleccionado y el comienzo y el final de la selección
selectAll(), select(int start, int end)	Selecciona todo o parte del texto
Métodos de TextField	**Función que realizan**
TextField(), TextField(int ncol), TextField(String s), TextField(String s, int ncol)	Constructores de TextField
int getColumns(), setColoumns(int)	Obtiene o establece el número de columnas del TextField
setEchoChar(char c), char getEchoChar(), boolean echoCharIsSet()	Establece, obtiene o pregunta por el carácter utilizado para passwords, de forma que no se pueda leer lo tecleado por el usuario
Métodos de TextArea	**Función que realizan**
TextArea(), TextArea(int nfil, int ncol), TextArea(String text), TextArea(String text, int nfil, int ncol)	Constructores de TextArea
setRows(int), setColumns(int), int getRows(), int getColumns()	Establecer y/u obtener los números de filas y de columnas
append(String str), insert(String str, int pos), replaceRange(String s, int i, int f)	Añadir texto al final, insertarlo en una posición determinada y reemplazar un texto determinado

Tabla 5.23. Métodos de las clases TextComponent, TextField y TextArea.

La clase ***TextComponent*** recibe eventos ***TextEvent***, y por lo tanto también los reciben sus clases derivadas ***TextField*** y ***TextAreas***. Este evento se produce cada vez que se modifica el texto del componente. La caja ***TextField*** soporta también el evento ***ActionEvent***, que se produce cada vez que el usuario termina de editar la única línea de texto pulsando ***Intro***.

Como es natural, las cajas de texto pueden recibir también los eventos de sus ***super-clases***, y más en concreto los eventos de ***Component***: ***FocusEvent***, ***MouseEvent*** y sobre todo ***KeyEvent***. Estos eventos permiten capturar las teclas pulsadas por el usuario y tomar las medidas adecuadas. Por ejemplo, si el usuario debe teclear un número en un ***TextField***, se puede crear una función que vaya capturando los caracteres tecleados y que rechace los que no sean numéricos.

Cuando se cambia desde programa el número de filas y de columnas de un ***TextField*** o ***TextArea***, hay que llamar al método ***validate()*** de la clase ***Component***, para que vuelva a aplicar el ***LayoutManager*** correspondiente. De todas formas, los tamaños fijados por el usuario tienen el carácter de "recomendaciones" o tamaños "preferidos", que el ***LayoutManager*** puede cambiar si es necesario.

5.2.26 Clase TextEvent

Se produce un ***TextEvent*** cada vez que cambia algo en un ***TextComponent*** (***TextArea*** y ***TextField***). Se puede desear evitar ciertos caracteres y para eso hay que gestionar los eventos correspondientes.

La interface ***TextListener*** tiene un único método: ***void textValueChanged(TextEvent te)***.

No hay métodos propios de la clase ***TextEvent***. Se puede utilizar el método ***Object getSource()***, que es heredado de ***EventObject***.

5.2.27 Clase KeyEvent

Se produce un ***KeyEvent*** al pulsar sobre el teclado. Como el teclado no es un componente del AWT, es el ***objeto que tiene el focus*** en ese momento quien genera los eventos ***KeyEvent*** (el objeto que es el *event source*).

Hay dos tipos de ***KeyEvents***:

1. ***key-typed***, que representa la introducción de un carácter Unicode.
2. ***key-pressed*** y ***key-released***, que representan pulsar o soltar una tecla. Son importantes para teclas que no representan caracteres, como por ejemplo F1.

Para estudiar estos eventos son muy importantes las ***Virtual KeyCodes*** (VKC). Las VKC son unas constantes que se corresponden con las ***teclas físicas*** del teclado, sin considerar minúsculas (que no tienen VKC). Se indican con el prefijo VK_, como VK_SHIFT o VK_A. La clase ***KeyEvent*** (en el package ***java.awt.event***) define constantes VKC para todas las teclas del teclado.

Por ejemplo, para escribir la letra "A" mayúscula se generan 5 eventos: *key-pressed VK_SHIFT*, *key-pressed VK_A*, *key-typed "A"*, *key-released VK_A*, y *key-released VK_SHIFT*. La Tabla 5.24 muestra algunos métodos de la clase ***KeyEvent*** y otros heredados de ***InputEvent***.

Métodos de la clase KeyEvent	**Función que realizan**
int getKeyChar()	Obtiene el carácter Unicode asociado con el evento
int getKeyCode()	Obtiene el VKC de la tecla pulsada o soltada
boolean isActionKey()	Indica si la tecla del evento es una ActionKey (HOME, END, …)
String getKeyText(int keyCode)	Devuelve un String que describe el VKC, tal como "HOME", "F1" o "A". Estos Strings se definen en el fichero awt.properties
String getKeyModifiersText(int modifiers)	Devuelve un String que describe las teclas modificadoras, tales como"Shift" o "Ctrl+Shift" (fichero awt.properties)
Métodos heredados de InputEvent	**Función que realizan**
boolean isShiftDown(), boolean isControlDown(), boolean isMetaDown(), boolean isAltDown(), int getModifiers()	Permiten identificar las teclas modificadoras

Tabla 5.24. Métodos de la clase KeyEvent.

Las constantes de ***InputEvent*** permiten identificar las teclas modificadoras y los botones del ratón. Estas constantes son SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK. A estas constantes se pueden aplicar las operaciones lógicas de bits para detectar combinaciones de teclas o pulsaciones múltiples.

5.3 MENUS

Los ***Menus*** de ***Java*** no descienden de ***Component***, sino de ***MenuComponent***, pero tienen un comportamiento similar, pues aceptan ***Events***. La Figura 5.4 muestra la jerarquía de clases de los ***Menus*** de ***Java 1.1***.

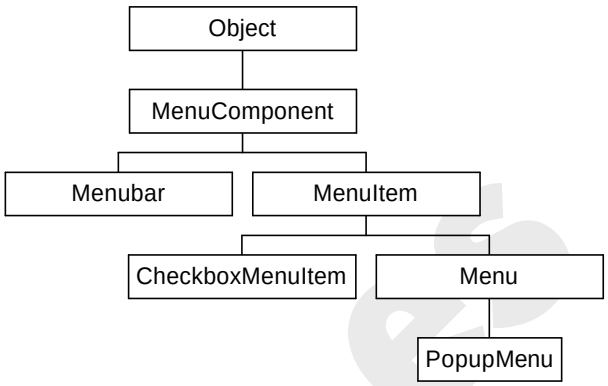


Figura 5.4. Jerarquía de clases para los menús.

Para crear un ***Menú*** se debe crear primero una ***MenuBar***; después se crean los ***Menus*** y los ***MenuItem***. Los ***MenuItems*** se añaden al ***Menu*** correspondiente; los ***Menus*** se añaden a la ***MenuBar*** y la ***MenuBar*** se añade a un ***Frame***. Una ***MenuBar*** se añade a un ***Frame*** con el método ***setMenuBar()***, de la clase ***Frame***. También puede añadirse un ***Menu*** a otro **Menu** para crear un ***sub-menú***, del modo que es habitual en ***Windows***.

La clase ***Menu*** es ***sub-clase*** de ***MenuItem***. Esto es así precisamente para permitir que un ***Menu*** sea añadido a otro ***Menu***.

5.3.1 Clase MenuShortcut

La clase ***java.awt.MenuShortcut*** (derivada de ***Object***) representa las teclas aceleradoras que pueden utilizarse para activar los menús desde teclado, sin ayuda del ratón. Se establece un ***Shortcut*** creando un objeto de esta clase con el constructor ***MenuShortcut(int vk_key)***, y pasándoselo al constructor adecuado de ***MenuItem***. Para más información, consúltese la adocumentación on-line de ***Java***. La Tabla 5.25 muestra algunos métodos de esta clase. Los ***MenuShortcut*** de ***Java*** están restringidos al uso la tecla control (CTRL). Al definir el ***MenuShortcut*** no hace falta incluir dicha tecla.

Métodos de la clase MenuShortcut	Función que realizan
MenuShortcut(int key)	Constructor
int getKey()	Obtiene el código virtual de la tecla utilizada como shortcut

Tabla 5.25. Métodos de la clase MenuShortcut.

5.3.2 Clase MenuBar

La Tabla 5.26 muestra algunos métodos de la clase ***MenuBar***. A una ***MenuBar*** sólo se pueden añadir objetos ***Menu***.

Métodos de MenuBar	Función que realizan
MenuBar()	Constructor
add(Menu), int getMenuCount(), Menu getMenu(int i)	Añade un manú, obtiene el número de menús y el menú en una posición determinada
MenuItem getShortcutMenuItem(MenuShortcut), deleteShortcut(MenuShortcut)	Obtiene el objeto MenuItem relacionado con un Shortcut y elimina el Shortcut especificado
remove(int index), remove(MenuComponent m)	Elimina un objeto Menu a partir de un índice o una referencia.
Enumeration shortcuts()	Obtiene un objeto Enumeration con todos los Shortcuts

Tabla 5.26. Métodos de la clase MenuBar.

5.3.3 Clase Menu

El objeto ***Menu*** define las opciones que aparecen al seleccionar uno de los menús de la barra de menús. En un ***Menu*** se pueden introducir objetos ***MenuItem***, otros objetos ***Menu*** (para crear sub-menús), objetos ***CheckboxMenuItem***, y ***separadores***. La Tabla 5.27 muestra algunos métodos de la clase ***Menu***. Obsérvese que como ***Menu*** desciende de ***MenuItem***, el método ***add(MenuItem)*** permite añadir objetos ***Menu*** a otro ***Menu***.

Métodos de Menu	**Función que realizan**
Menu(String)	Constructor a partir de una etiqueta
int getItemCount()	Obtener el número de items
MenuItem getItem(int)	Obtener el MenuItem a partir de un índice
add(String), add(MenuItem), addSeparator(), insertSeparator(int index)	Añadir un MenuItem o un separador
remove(int index), remove(MenuComponent), removeAll()	Eliminar uno o todos los componentes
insert(String lbl, int index), insert(MenuItem mnu, int index)	Insertar items en una posición dada
String getLabel(), setLabel(String)	Obtener y establecer las etiquetas de los items

Tabla 5.27. Métodos de la clase Menu.

5.3.4 Clase MenuItem

Los objetos de la clase ***MenuItem*** representan las distintas opciones de un menú. Al seleccionar, en la ejecución del programa, un objeto ***MenuItem*** se generan eventos del tipo ***ActionEvents***. Para cada item de un ***Menu*** se puede definir un ***ActionListener***, que define el método ***actionPerformed()***. La Tabla 5.28 muestra algunos métodos de la clase ***MenuItem***.

Métodos de MenuItem	**Función que realizan**
MenuItem(String lbl), MenuItem(String, MenuShortcut)	Constructores. El carácter (-) es el label de los separators
boolean isEnabled(), setEnabled(boolean)	Pregunta y determina si en item está activo
String getLabel(), setLabel(String)	Obtiene y establece la etiqueta del item
MenuShortcut getShortcut(), setShortcut(MenuShortcut), deleteShortcut(MenuShortcut)	Permiten obtener, establecer y borrar los MenuShortcuts
String getActionCommand(), setActionCommand(String)	Para obtener y establecer un identificador distinto del label

Tabla 5.28. Métodos de la clase MenuItem.

El método ***getActionCommand()***, asociado al ***getSource()*** del evento correspondiente, no permite identificar correctamente al ***item*** cuando éste se ha activado mediante el ***MenuShortcut*** (en ese caso devuelve ***null***).

5.3.5 Clase CheckboxMenuItem

Son ***items*** de un ***Menu*** que pueden estar activados o no activados. La clase ***CheckboxMenuItem*** no genera un ***ActionEvent***, sino un ***ItemEvent***, de modo similar a la clase ***Checkbox*** (ver Apartado 5.2.17 en la página 103). En este caso hará registrar un ***ItemListener***. La Tabla 5.29 muestra algunos métodos de esta clase.

Métodos de la clase CheckboxMenuItem	Función que realizan
CheckboxMenuItem(String lbl), CheckboxMenuItem(String lbl, boolean state)	Constructores
boolean getState(), setState(boolean)	Permiten obtener y establecer el estado del ChecboxMenuItem

Tabla 5.29. Métodos de la clase CheckboxMenuItem.

5.3.6 Menús pop-up

Los menús ***pop-up*** son menús que aparecen en cualquier parte de la pantalla al clicar con el botón derecho del ratón (***pop-up trigger***) sobre un componente determinado (***parent Component***). El menú ***pop-up*** se muestra en unas coordenadas relativas al ***parent Component***, que debe estar visible.

Métodos de PopupMenu	Función que realizan
PopupMenu(), PopupMenu(String title)	Constructores de PopupMenu:
show(Component origin, int x, int y)	Muestra el pop-up menú en la posición indicada

Tabla 5.30. Métodos de la clase PopupMenu.

Además, se pueden utilizar los métodos de la clase ***Menu*** (ver página 111), de la que deriva ***PopupMenu***. Para hacer que aparezca el ***PopupMenu*** habrá que registrar el ***MouseListener*** y definir el método ***mouseClicked()***.

5.4 LAYOUT MANAGERS

La portabilidad de ***Java*** a distintas plataformas y distintos sistemas operativos necesita flexibilidad a la hora de situar los ***Components*** (*Buttons*, *Canvas*, *TextAreas*, etc.) en un ***Container*** (*Window*, *Panel*, …). Un ***Layout Manager*** es un objeto que controla cómo los ***Components*** se sitúan en un ***Container***.

5.4.1 Concepto y Ejemplos de LayoutsManagers

El AWT define cinco ***Layout Managers***: dos muy sencillos (***FlowLayout*** y ***GridLayout***), dos más especializados (***BorderLayout*** y ***CardLayout***) y uno muy general (***GridBagLayout***). Además, los usuarios pueden escribir su propio ***Layout Manager***, implementando la interface ***LayoutManager***, que especifica 5 métodos. ***Java*** permite también posicionar los ***Components*** de ***modo absoluto***, sin ***Layout Manager***, pero de ordinario puede perderse la portabilidad y algunas otras características.

Todos los ***Containers*** tienen un ***Layout Manager*** por defecto, que se utiliza si no se indica otra cosa: Para ***Panel***, el defecto es un objeto de la clase ***FlowLayout***. Para ***Window*** (***Frame*** y ***Dialog***), el defecto es un objeto de la clase ***BorderLayout***.

La Figura 5.5 muestra un ejemplo de ***FlowLayout***: Los componentes se van añadiendo de izda a dcha y de arriba hacia abajo. Se puede elegir alineación por la izquierda, centrada o por la derecha, respecto al container.

La Figura 5.6 muestra un ejemplo de ***BorderLayout***: el container se divide en 5 zonas: ***North***, ***South***, ***East***, ***West*** y ***Center*** (que ocupa el resto de espacio).

El ejemplo de ***GridLayout*** se muestra en la Figura 5.7. Se utiliza una ***matriz de celdas*** que se numeran como se muestra en dicha figura (de izda a dcha y de arriba a abajo).

La Figura 5.8 muestra un ejemplo de uso del ***GridBagLayout***. Se utiliza también una matriz de celdas, pero permitiendo que algunos componentes ocupen más de una celda.

Finalmente, la Figura 5.9 y las dos Figuras siguientes muestran un ejemplo de ***CardLayout***. En este caso se permite que el mismo espacio sea utilizado sucesivamente por contenidos diferentes.

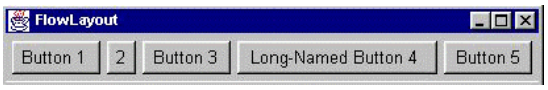


Figura 5.5. Ejemplo de FlowLayout.

Figura 5.6. Ejemplo de BorderLayout

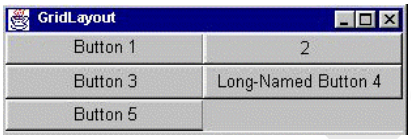


Figura 5.7. Ejemplo de GridLayout.

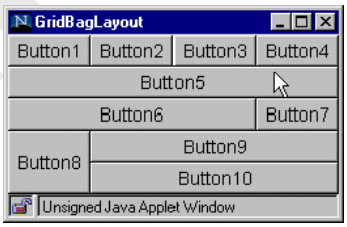


Figura 5.8. Ejemplo de GridBagLayout.

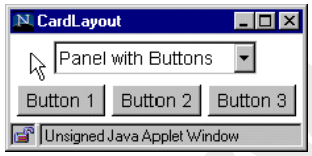


Figura 5.9. CardLayout: pantalla 1.

Figura 5.10. CardLayout: pantalla 2

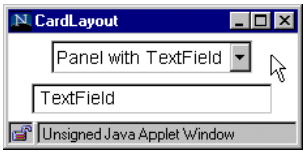


Figura 5.11. CardLayout: pantalla 3.

5.4.2 Ideas generales sobre los LayoutManagers

Se debe elegir el ***Layout Manager*** que mejor se adecúe a las necesidades de la aplicación que se desea desarrollar. Recuérdese que cada ***Container*** tiene un ***Layout Manager*** por defecto. Si se desea utilizar el ***Layout Manager*** por defecto basta crear el ***Container*** (su constructor crea un objeto del ***Layout Manager*** por defecto e inicializa el ***Container*** para hacer uso de él).

Para utilizar un ***Layout Manager*** diferente hay que crear un objeto de dicho ***Layout Manager*** y pasárselo al constructor del container o decirle a dicho container que lo utilice por medio del método ***setLayout()***, en la forma:

```
unContainer.setLayout(new GridLayout());
```

La clase ***Container*** dispone de métodos para manejar el ***Layout Manager*** (ver Tabla 5.31):

Si se cambia de modo indirecto el tamaño de un ***Component*** (por ejemplo cambiando el tamaño del ***Font***), hay que llamar al método ***invalidate()*** del ***Component*** y luego al método ***validate()*** del ***Container***, lo que hace que se ejecute el método ***doLayout()*** para reajustar el espacio disponible.

Métodos de Container para manejar Layout Managers	Función que realizan
add()	Permite añadir Components a un Container
remove() y removeAll()	Permiten eliminar Components de un Container
doLayout(), validate()	doLayout() se llama automáticamente cada vez que hay que redibujar el Container y sus Components. Se llama también cuando el usuario llama al método validate()

Tabla 5.31. Métodos de Container para manejar los Layout Managers.

5.4.3 FlowLayout

FlowLayout es el ***Layout Manager*** por defecto para ***Panel***. ***FlowLayout*** coloca los componentes uno detrás deotro, en una fila, de izda a dcha y de arriba a abajo, en la misma forma en que procede un procesador de texto con las palabras de un párrafo. Los componentes se añaden en el mismo orden en que se ejecutan los métodos ***add()***. Si se cambia el tamaño de la ventana los componentes se redistribuyen de modo acorde, ocupando más filas si es necesario.

La clase ***FlowLayout*** tiene tres constructores:

```
FlowLayout();
FlowLayout(int alignement);
FlowLayout(int alignement, int horizontalGap, int verticalGap);
```

Se puede establecer la alineación de los componentes (centrados, por defecto), por medio de las constantes FlowLayout.LEFT, FlowLayout.CENTER y FlowLayout.RIGHT.

Es posible también establecer una distancia horizontal y vertical entre componentes (el ***gap***, en pixels). El valor por defecto son 5 pixels.

5.4.4 BorderLayout

BorderLayout es el ***Layout Manager*** por defecto para ***Windows*** y ***Frames***. ***BorderLayout*** define cinco áreas: ***North***, ***South***, ***East***, ***West*** y ***Center***. Si se aumenta el tamaño de la ventana todas las zonas se mantienen en su mínimo tamaño posible excepto ***Center***, que absorbe casi todo el crecimiento. Los componentes añadidos en cada zona tratan de ocupar todo el espacio disponible. Por ejemplo, si se añade un botón, el botón se hará tan grande como la celda, lo cual puede producir efectos muy extraños. Para evitar esto se puede introducir en la celda un panel con ***FlowLayout*** y añadir el botón al panel y el panel a la celda.

Los constructores de ***BorderLayout*** son los siguientes:

```
BorderLayout();
BorderLayout(int horizontalGap, int verticalGap);
```

Por defecto ***BorderLayout*** no deja espacio entre componentes. Al añadir un componente a un ***Container*** con ***BorderLayout*** se debe especificar la zona como segundo argumento:

```
miContainer.add(new Button("Norte"), "North");
```

5.4.5 GridLayout

Con ***GridLayout*** las componentes se colocan en una matriz de celdas. Todas las celdas tienen el mismo tamaño. Cada componente utiliza todo el espacio disponible en su celda, al igual que en ***BorderLayout***.

GridLayout tiene dos constructores:

```
GridLayout(int nfil, int ncol);
GridLayout(int nfil, int ncol, int horizontalGap, int verticalGap);
```

Al menos uno de los parámetros ***nfil*** y ***ncol*** debe ser distinto de cero. El valor por defecto para el espacio entre filas y columnas es cero pixels.

5.4.6 CardLayout

CardLayout permite disponer distintos componentes (de ordinario ***Panels***) que comparten la misma ventana para ser mostrados sucesivamente. Son como transparencias, diapositivas o cartas de baraja que van apareciendo una detrás de otra.

El ***orden*** de las "cartas" se puede establecer de los siguientes modos:

1. Yendo a la primera o a la última, de acuerdo con el orden en que fueron añadidas al container.
2. Recorriendo las cartas hacia delante o hacia atrás, de una en una.
3. Mostrando una carta con un nombre determinado.

Los ***constructores*** de esta clase son:

```
CardLayout()
CardLayout(int horizGap, int vertGap)
```

Para añadir componentes a un container con ***CardLayout*** se utiliza el método:

```
Container.add(Component comp, int index)
```

donde ***index*** indica la posición en que hay que insertar la carta. Los siguientes métodos de ***CardLayout*** permiten controlar el orden en que aparecen las cartas:

```
void first(Container cont);
void last(Container cont);
void previous(Container cont);
void next(Container cont);
void show(Container cont, String nameCard);
```

5.4.7 GridBagLayout

El ***GridBagLayout*** es el ***Layout Manager*** más completo y flexible, aunque también el más complicado de entender y de manejar. Al igual que el ***GridLayout***, el ***GridBagLayout*** parte de una matriz de celdas en la que se sitúan los componentes. La diferencia está en que las filas pueden tener distinta altura, las columnas pueden tener distinta anchura, y además en el ***GridBagLayout*** un componente puede ocupar varias celdas contiguas.

La posición y el tamaño de cada componente se especifican por medio de unas “restricciones” o ***constraints***. Las restricciones se establecen creando un objeto de la clase ***GridBagConstraints***,

dando valor a sus propiedades (variables miembro) y asociando ese objeto con el componente por medio del método ***setConstraints()***.

Las ***variables miembro*** de ***GridBagConstraints*** son las siguientes:

- ***gridx*** y ***gridy***. Especifican la fila y la columna en la que situar la esquina superior izquierda del componente (se empieza a contar de cero). Con la constante GridBagConstraints.RELATIVE se indica que el componente se sitúa relativamente al anterior componente situado (es la condición por defecto).
- ***gridwidth*** y ***gridheight***. Determinan el número de columnas y de filas que va a ocupar el componente. El valor por defecto es una columna y una fila. La constante GridBagConstraints.REMAINDER indica que el componente es el último de la columna o de la fila, mientras que GridBagConstraints.RELATIVE indica que el componente se sitúa respecto al anterior componente de la fila o columna.
- ***fill***. En el caso en que el componente sea más pequeño que el espacio reservado, esta variable indica si debe ocupar o no todo el espacio disponible. Los posibles valores son: GridBagConstraints.NONE (no lo ocupa; defecto), GridBagConstraints.HORIZONTAL (lo ocupa en dirección horizontal), GridBagConstraints.VERTICAL (lo ocupa en vertical) y GridBagConstraints.BOTH (lo ocupa en ambas direcciones).
- ***ipadx*** y ***ipady***. Especifican el espacio a añadir en cada dirección al tamaño interno del componente. Los valores por defecto son cero. El tamaño del componente será el tamaño mínimo más dos veces el ***ipadx*** o el ***ipady***.
- ***insets***. Indican el espacio mínimo entre el componente y el espacio disponible. Se establece con un objeto de la clase ***java.awt.Insets***. Por defecto es cero.
- ***anchor***. Se utiliza para determinar dónde se coloca el componente, cuando éste es menor que el espacio disponible. Sus posibles valores vienen dados por las constantes de la clase ***GridBagConstraints***: CENTER (el valor por defecto), NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST y NORTHWEST.
- ***weightx*** y ***weighty***. Son unos coeficientes entre 0.0 y 1.0 que sirven para dar más o menos "peso" a las distintas filas y columnas. A más "peso" más probabilidades tienen de que se les dé más anchura o más altura.

A continuación se muestra una forma típica de crear un container con ***GridBagLayout*** y de añadirle componentes:

```
GridBagLayout unGBL = new GridBagLayout();
GridBagConstraints unasConstr = new GridBagConstraints();
unContainer.setLayout(unGBL);

// Ahora ya se pueden añadir los componentes
//...Se crea un componente unComp
//...Se da valor a las variables del objeto unasConstr
// Se asocian las restricciones con el componente
unGBL.setConstraints(unComp, unasConstr);
// Se añade el componente al container
unContainer.add(unComp);
```

5.5 GRÁFICOS, TEXTO E IMÁGENES

En esta parte final del AWT se van a describir, también muy sucintamente, algunas clases y métodos para realizar dibujos y añadir texto e imágenes a la interface gráfica de usuario.

5.5.1 Capacidades gráficas del AWT: Métodos paint(), repaint() y update()

La clase ***Component*** tiene tres métodos muy importantes relacionados con gráficos: ***paint()***, ***repaint()*** y ***update()***. Cuando el usuario llama al método ***repaint()*** de un componente, el AWT llama al método ***update()*** de ese componente, que por defecto llama al método ***paint()***.

5.5.1.1 Método paint(Graphics g)

El método ***paint()*** está definido en la clase ***Component***, pero ese método no hace nada y hay que redefinirlo en una de sus clases derivadas. El programador no tiene que preocuparse de llamar a este método: el sistema operativo lo llama al dibujar por primera vez una ventana, y luego lo vuelve a llamar cada vez que entiende que la ventana o una parte de la ventana debe ser re-dibujada (por ejemplo, por haber estado tapada por otra ventana y quedar de nuevo a la vista).

5.5.1.2 Método update(Graphics g)

El método ***update()*** hace dos cosas: primero re-dibuja la ventana con el color de fondo y luego llama al método ***paint()***. Este método también es llamado por el AWT, y también puede ser llamado por el programador, quizás porque ha realizado algún cambio en la ventana y necesita que se dibuje de nuevo.

La propia estructura de este método -el comenzar pintando de nuevo con el color de fondo- hace que se produzca ***parpadeo*** (***flicker***) en las animaciones. Una de las formas de evitar este efecto es redefinir este método de una forma diferente, cambiando de una imagen a otra sólo lo que haya que cambiar, en vez de re-dibujar todo otra vez desde el principio. Este método no siempre proporciona los resultados buscados y hay que recurrrir al método del ***doble buffer***.

5.5.1.3 Método repaint()

Este es el método que con más frecuencia es llamado por el programador. El método ***repaint()*** llama "lo antes posible" al método ***update()*** del componente. Se puede también especificar un número de milisegundos para que el método ***update()*** se llame transcurrido ese tiempo. El método ***repaint()*** tiene las cuatro formas siguientes:

```
repaint()
repaint(long time)
repaint(int x, int y, int w, int h)
repaint(long time, int x, int y, int w, int h)
```

Las formas tercera y cuarta permiten definir una ***zona rectangular*** de la ventana a la que aplicar el método.

5.5.2 Clase Graphics

El único argumento de los métodos ***update()*** y ***paint()*** es un objeto de esta clase. La clase ***Graphics*** dispone de métodos para soportar dos tipos de gráficos:

1. Dibujo de ***primitivas gráficas*** (*texto, líneas, círculos, rectángulos, polígonos, …*).
2. Presentación de ***imágenes*** en formatos ****.gif*** y ****.jpeg***.

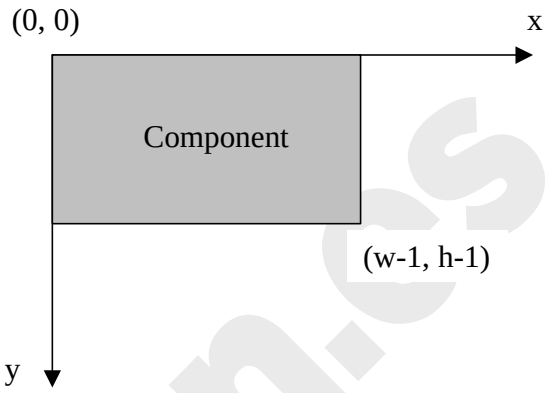


Figura 5.12. Coordenadas de los gráficos de Java.

Además, la clase ***Graphics*** mantiene un ***contexto gráfico***: un área de dibujo actual, un color de dibujo del background y otro del foreground, un font con todas sus propiedades, etc. La Figura 5.12 muestra el sistema de coordenadas utilizado en ***Java***. Como es habitual en Informática, los ejes están situados en la esquina superior izquierda, con la orientación indicada en la Figura 5.12 (eje de ordenadas descendente). Las coordenadas se miden siempre en ***pixels***.

5.5.3 Primitivas gráficas

Java dispone de métodos para realizar dibujos sencillos, llamados a veces "primitivas" gráficas. Como se ha dicho, las coordenadas se miden en ***pixels***, empezando a contar desde cero. La clase ***Graphics*** dispone de los métodos para primitivas gráficas reseñados en la Tabla 5.32.

Excepto los polígonos y las líneas, todas las formas geométricas se determinan por el rectángulo que las comprende, cuyas dimensiones son ***w*** y ***h***. Los polígonos admiten un argumento de la clase ***java.awt.Polygon***.

Método gráfico	**Función que realizan**
drawLine(int x1, int y1, int x2, int y2)	Dibuja una línea entre dos puntos
drawRect(int x1, int y1, int w, int h)	Dibuja un rectángulo (w-1, h-1)
fillRect(int x1, int y1, int w, int h)	Dibuja un rectángulo y lo rellena con el color actual
clearRect(int x1, int y1, int w, int h)	Borra dibujando con el background color
draw3DRect(int x1, int y1, int w, int h, boolean raised)	Dibuja un rectángulo resaltado (w+1, h+1)
fill3DRect(int x1, int y1, int w, int h, boolean raised)	Rellena un rectángulo resaltado (w+1, h+1)
drawRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Dibuja un rectángulo redondeado
fillRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Rellena un rectángulo redondeado
drawOval(int x1, int y1, int w, int h)	Dibuja una elipse
fillOval(int x1, int y1, int w, int h)	Dibuja una elipse y la rellena de un color
drawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Dibuja un arco de elipse (ángulos en grados)
fillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Rellena un arco de elipse
drawPolygon(int x[], int y[], int nPoints)	Dibuja y cierra el polígono de modo automático
drawPolyline(int x[], int y[], int nPoints)	Dibuja un polígono pero no lo cierra
fillPolygon(int x[], int y[], int nPoints)	Rellena un polígono

Tabla 5.32. Métodos de la clase Graphics para dibujo de primitivas gráficas.

Los métodos ***draw3DRect()***, ***fill3DRect()***, ***drawOval()***, ***fillOval()***, ***drawArc()*** y ***fillArc()*** dibujan objetos cuyo tamaño total es (w+1, h+1) pixels.

5.5.4 Clases Graphics y Font

La clase ***Graphics*** permite "dibujar" texto, como alternativa al texto mostrado en los componentes ***Label***, ***TextField*** y ***TextArea***. Los métodos de esta clase para dibujar texto son los siguientes:

```
drawBytes(byte data[], int offset, int length, int x, int y);
drawChars(char data[], int offset, int length, int x, int y);
drawString(String str, int x, int y);
```

En estos métodos, los argumentos ***x*** e ***y*** representan las coordenadas de la ***línea base*** (ver Figura 5.13). El argumento ***offset*** indica el elemento del array que se empieza a imprimir.

Cada tipo de letra está representado por un objeto de la clase ***Font***. Las clases ***Component*** y ***Graphics*** disponen de métodos ***setFont()*** y ***getFont()***. El constructor de ***Font*** tiene la forma:

```
Font(String name, int style, int size)
```

donde el ***style*** se puede definir con las constantes Font.PLAIN, Font.BOLD y Font.ITALIC. Estas constantes se pueden combinar en la forma: Font.BOLD | Font.ITALIC.

La clase ***Font*** tiene tres variables *protected*, llamadas ***name***, ***style*** y ***size***. Además tiene tres constantes enteras: PLAIN, BOLD e ITALIC. Esta clase dispone de los métodos ***String getName()***, ***int getStyle()***, ***int getSize()***, ***boolean isPlain()***, ***boolean isBold()*** y ***boolean isItalic()***, cuyo significado es inmediato.

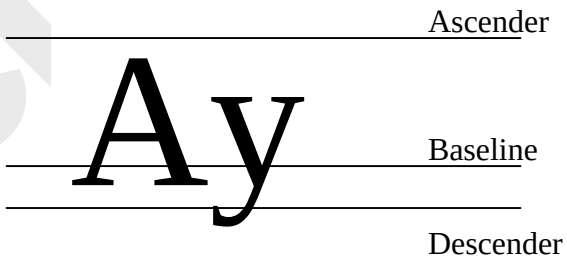


Figura 5.13. Líneas importantes en un tipo de letra.

Para mayor portabilidad se recomienda utilizar nombres lógicos de fonts, tales como ***Serif*** (*Times New Roman*), ***SansSerif*** (*Arial*) y ***Monospaced*** (*Courier*).

5.5.5 Clase FontMetrics

La clase ***FontMetrics*** permite obtener información sobre una ***font*** y sobre el espacio que ocupa un ***char*** o un ***String*** utilizando esa ***font***. Esto es muy útil cuando se pretende rotular algo de modo que quede siempre centrado y bien dimensionado.

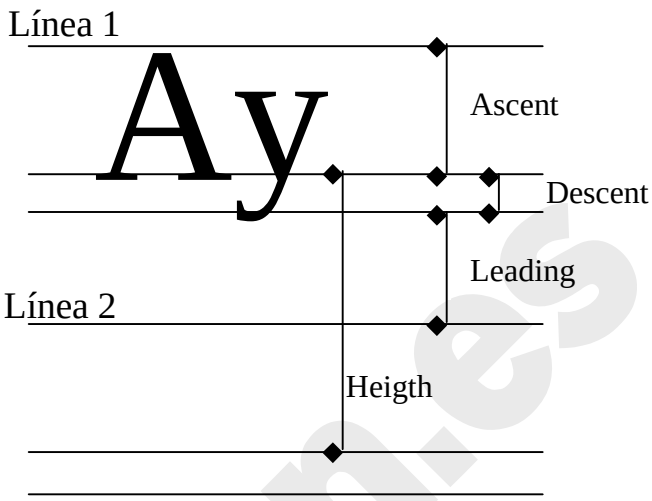


Figura 5.14. Nomenclatura para la clase FontMetrics.

La clase ***FontMetrics*** es una clase ***abstract***. Esto quiere decir que no se pueden crear directamente objetos de esta clase ni llamar a su constuctor. La forma habitual de soslayar esta dificultad es creando una subclase. En la práctica ***Java*** resuelve esta dificultad para el usuario, ya que la clase ***FontMetrics*** tiene como variable miembro un objeto de la clase ***Font***. Por ello, un objeto de la clase ***FontMetrics*** contiene información sobre la ***font*** que se le ha pasado como argumento al constructor. De todas formas, el camino más habitual para obtener esa información es a partir de un objeto de la clase ***Graphics*** que ya tiene un ***font*** definido. A partir de un objeto ***g*** de la clase ***Graphics*** se puede obtener una referencia ***FontMetrics*** en la forma:

```
FontMetrics miFontMet = g.getFontMetrics();
```

donde está claro que se está utilizando una referencia de la clase *abstract* ***FontMetrics*** para refererirse a un objeto de una clase derivada creada dentro del API de ***Java***. Con una referencia de tipo ***FontMetrics*** se pueden utilizar todos los métodos propios de dicha clase.

La Tabla 5.33 muestra algunos métodos de la clase ***FontMetrics***, para los que se debe tener en cuenta la terminología introducida en la Figura 5.14.

Métodos de la clase FontMetrics	**Función que realizan**
FontMetrics(Font font)	Constructor
int getAscent(), int getMaxAscent()	Permiten obtener el “Ascent” actual y máximo para esa font
int getDescent(), int getMaxDescent()	Permiten obtener el “Descent” actual y máximo para esa font
int getHeight(), int getLeading()	Permiten ontener la distancia entre líneas y la distancia entre el descender de una línea y el ascender de la siguiente
int getMaxAdvance()	Da la máxima anchura de un carácter de esa font, incluyendo el espacio hasta el siguiente carácter
int charWidth(int ch), int charWidth(char ch), int stringWidth(String str)	Dan la anchura de un carácter (incluyendo el espacio hasta el siguiente carácter) o de toda una cadena de caracteres
int charsWidth(char data[], int start, int len), int bytesWidth(byte data[], int start, int len)	Dan la anchura de un array de caracteres o de bytes. Permiten definir la posición del comienzo y el número de caracteres

Tabla 5.33. Métodos de la clase FontMetrics.

5.5.6 Clase Color

La clase ***java.awt.Color*** encapsula colores utilizando el formato RGB (Red, Green, Blue). Las componentes de cada color primario en el color resultante se expresan con números enteros entre 0 y 255, siendo 0 la intensidad mínima de ese color, y 255 la máxima.

En la clase ***Color*** existen constantes para colores predeterminados de uso frecuente: ***black***, ***white***, ***green***, ***blue***, ***red***, ***yellow***, ***magenta***, ***cyan***, ***orange***, ***pink***, ***gray***, ***darkGray***, ***lightGray***. La Tabla 5.34 muestra algunos métodos de la clase ***Color***.

Metodos de la clase Color	**Función que realizan**
Color(int), Color(int,int,int), Color(float,float,float)	Constructores de Color, con tres bytes en un int (del bit 0 al 23), con enteros entre 0 y 255 y float entre 0.0 y 1.0
Color brighter(), Color darker()	Obtienen una versión más o menos brillante de un color
Color getColor(), int getRGB()	Obtiene un color en los tres primeros bytes de un int
int getGreen(), int getRed(), int getBlue()	Obtienen las componentes de un color
Color getHSBColor()	Obtiene un color a partir de los valores de “hue”, “saturation” y “brightness” (entre 0.0 y 1.0)
float[] RGBtoHSB(int,int,int,float[]), int HSBtoRGB(float,float,float)	Métodos static para convertir colores de un sistema de definición de colores a otro

Tabla 5.34. Métodos de la clase Color.

5.5.7 Imágenes

Java permite incorporar imágenes de tipo GIF y JPEG definidas en ficheros. Se dispone para ello de la clase ***java.awt.Image***. Para cargar una imagen hay que indicar la localización del fichero (URL) y cargarlo mediante los métodos ***Image getImage(String)*** o ***Image getImage(URL, String)***. Estos métodos existen en las clases ***java.awt.Toolkit*** y ***java.applet.Applet***. El argumento de tipo ***String*** representa una variable conteniendo el nombre del fichero.

Cuando estas imágenes se cargan en ***applets***, para obtener el URL pueden ser útiles las funciones ***getDocumentBase()*** y ***getCodeBase()***, que devuelven el URL del fichero HTML que llama al ***applet***, y el directorio que contiene el ***applet*** (en forma de ***String***).

Para cargar una imagen hay que comenzar creando un objeto ***Image***, y llamar al método ***getImage()***, pasándole como argumento el URL. Por ejemplo:

```
Image miImagen = getImage(getCodeBase(), "imagen.gif")
```

Una vez cargada la imagen, hay que representarla, para lo cual se redefine el método ***paint()*** para llamar al método ***drawImage()*** de la clase ***Graphics***. Dicho método admite varias formas, aunque casi siempre hay que incluir el nombre del objeto imagen creado, las dimensiones de dicha imagen y un objeto ***ImageObserver***.

ImageObserver es una interface que declara métodos para observar el estado de la carga y visualización de la imagen. Si se está programando un ***applet***, basta con poner como ***ImageObserver*** la referencia ***this***, ya que en la mayoría de los casos, la implementación de esta

interface en la clase ***Applet*** proporciona el comportamiento deseado. Para más información sobre dicho método dirigirse a la referencia de la API.

La clase ***Image*** define ciertas constantes para controlar los algoritmos de cambio de escala:

SCALE_DEFAULT, SCALE_FAST, SCALE_SMOOTH,

SCALE_REPLICATE, SCALE_AVERAGE.

La Tabla 5.35 muestra algunos métodos de la clase ***Image***.

Métodos de la clase Image	**Función que realizan**
Image()	Constructor
int getWidth(ImageObserver) int getHeight(ImageObserver)	Determinan la anchura y la altura de la imagen. Si no se conocen todavía, este método devuelve -1 y el objeto ImageObserver especificado será notificado más tarde
Graphics getGraphics()	Crea un contexto gráfico para poder dibujar en una imagen no visible en pantalla. Este método sólo se puede llamar para objetos no visibles en pantalla
Object getProperty(String, ImageObserver)	Obtiene una propiedad de una imagen a partir del nombre de la propiedad
Image getScaledInstance(int w, int h, int hints)	Crea una versión de la imagen a otra escala. Si w o h son negativas se utiliza la otra dimensión manteniendo la proporción. El último argumento es información para el algoritmo de cambio de escala

Tabla 5.35. Métodos de la clase Image.

5.6 ANIMACIONES

Las animaciones tienen un gran interés desde diversos puntos de vista. Una imagen vale más que mil palabras y una imagen en movimiento es todavía mucho más útil. Para presentar o describir ciertos conceptos el movimiento animado es fundamental. Además, las animaciones o mejor dicho, la forma de hacer animaciones en ***Java*** ilustran mucho la forma en que dicho lenguaje realiza los gráficos. En estos apartados se va a seguir el esquema del ***Tutorial*** de ***Sun*** sobre el AWT.

Se pueden hacer animaciones de una forma muy sencilla: se define el método ***paint()*** de forma que cada vez que sea llamado dibuje algo diferente de lo que ha dibujado la vez anterior. De todas formas, recuérdese que el programador no llama directamente a este método. El programador llama al método ***repaint()***, quizás dentro de un bucle ***while*** que incluya una llamada al método ***sleep()*** de la clase ***Thread*** (ver Capítulo 6, a partir de la página 125), para esperar un cierto número de milisegundos entre dibujo y dibujo (entre *frame* y *frame*, utilizando la terminología de las animaciones). Recuérdese que ***repaint()*** llama a ***update()*** lo antes posible, y que ***update()*** borra todo redibujando con el color de fondo y llama a ***paint()***.

La forma de proceder descrita da buenos resultados para animaciones muy sencillas, pero produce ***parpadeo*** o ***flicker*** cuando los gráficos son un poco más complicados. La razón está en el propio proceso descrito anteriormente, combinado con la velocidad de refresco del monitor. La velocidad de refresco vertical de un monitor suele estar entre 60 y 75 herzios. Eso quiere decir que la imagen se actualiza unas 60 ó 75 veces por segundo. Cuando el refresco se realiza después de haber borrado la imagen anterior pintando con el color de fondo y antes de que se termine de dibujar de nuevo toda la imagen, se obtiene una imagen incompleta, que sólo aparecerá terminada en uno de

los siguientes pasos de refresco del monitor. Ésta es la causa del ***flicker***. A continuación se verán dos formas de reducirlo o eliminarlo.

5.6.1 Eliminación del parpadeo o flicker redefiniendo el método update()

El problema del ***flicker*** se localiza en la llamada al método ***update()***, que borra todo pintando con el color de fondo y después llama a ***paint()***. Una forma de resolver esta dificultad es ***re-definir*** el método ***update()***, de forma que se adapte mejor al problema que se trata de resolver.

Una posibilidad es no re-pintar todo con el color de fondo, no llamar a ***paint()*** e introducir en ***update()*** el código encargado de realizar los dibujos, cambiando sólo aquello que haya que cambiar. A pesar de esto, es necesario re-definir ***paint()*** pues es el método que se llama de forma automática cuando la ventana de ***Java*** es tapada por otra que luego se retira. Una posible solución es hacer que ***paint()*** llame a ***update()***, terminando por establecer un orden de llamadas opuesto al de defecto. Hay que tener en cuenta que, al no borrar todo pintando con el color de fondo, el programador tiene que preocuparse de borrar de forma selectiva entre *frame* y *frame* lo que sea necesario. Los métodos ***setClip()*** y ***clipRect()*** de la clase ***Graphics*** permiten hacer que las operaciones gráficas no surtan efecto fuera de un área rectangular previamente determinada. Al ser dependiente del tipo de gráficos concretos de que se trate, este método no siempre proporciona soluciones adecuadas.

5.6.2 Técnica del doble buffer

La técnica del ***doble buffer*** proporciona la mejor solución para el problema de las animaciones, aunque requiere una programación algo más complicada. La idea básica del ***doble buffer*** es realizar los dibujos en una imagen invisible, distinta de la que se está viendo en la pantalla, y hacerla visible cuando se ha terminado de dibujar, de forma que aparezca instantáneamente.

Para crear el segundo buffer o imagen invisible hay que crear un objeto de la clase ***Image*** del mismo tamaño que la imagen que se está viendo y crear un contexto gráfico u objeto de la clase ***Graphics*** que permita dibujar sobre la imagen invisible. Esto se hace con las sentencias,

```
Image imgInv;
Graphics graphInv;
Dimension dimInv;
Dimension d = size(); // se obtiene la dimensión del panel
```

en la clase que controle el dibujo (por ejemplo en una clase que derive de ***Panel***). El siguiente paso es ***re-definir*** el método ***update()*** de forma que no borre la imagen anterior con el color de fondo. Deberá llamar directamente al método ***paint()***:

```
public void update (Graphics g) {
   paint(g); // Sin borrar la imagen anterior se llama al método paint()
} // fin del método update()
```

En el método ***paint()*** se modifica el código de modo que primero se dibuje en la imagen invisible y luego ésta se haga visible:

```
public void paint (Graphics g) {
   // se comprueba si existe el objeto invisible y si sus dimensiones son correctas
   if ((graphInv==null) || (d.width!=dimInv.width) || (d.height!=dimInv.height)) {
       dimInv = d;
       // se llama al método createImage de la clase Component
       imgInv = createImage(d.width, d.height);
       // se llama al método getGraphics de la clase Image
       graphInv = imgInv.getGraphics();
   }

   // se establecen las propiedades del contexto gráfico invisible,
   //    y se dibuja sobre él
   graphInv.setColor(Color.white);  // Se borra pintando de blanco
   graphInv.fillRect(0, 0, dimInv.width, dimInv.height);
   graphInv.setColor(Color.white);  // Se asigna un color para dibujar
   // Sentencias para dibujar sobre graphInv
   graphInv.drawLine(0, 0, 100, 100);
   ...
   // finalmente se hace visible la imagen invisible a partir del punto (0, 0)
   //     utilizando el propio panel como ImageObserver
   g.drawImage(imgInv, 0, 0, this);
} // fin del método paint()
```

Los gráficos y las animaciones son particularmente útiles en las applets. El ***Tutorial*** de ***Sun*** tiene un ejemplo (un ***applet***) completamente explicado y desarrollado sobre las animaciones y los distintos métodos de eliminar el flicker o parpadeo.

6. THREADS: PROGRAMAS MULTITAREA

Los procesadores y los Sistemas Operativos modernos permiten la ***multitarea***, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, un ordenador con una sola CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En ordenadores con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un ***hilo*** o ***thread*** diferente. La Figura 6.1, tomada del ***Tutorial*** de ***Sun***, muestra los esquemas correspondientes a un programa con una o dos ***threads***.

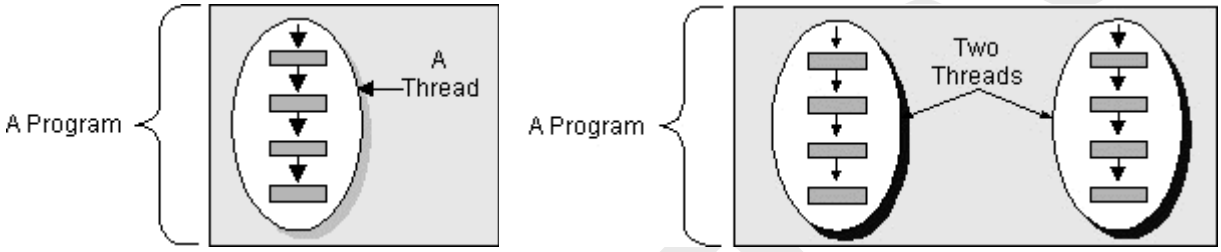


Figura 6.1. Programa con 1 y con 2 threads o hilos.

Un ***proceso*** es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un ***proceso*** simultáneamente. Un ***thread*** o ***hilo*** es un ***flujo secuencial simple*** dentro de un ***proceso***. Un único ***proceso*** puede tener varios ***hilos*** ejecutándose. Por ejemplo el programa ***Netscape*** sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un ***hilo***.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de ***threads*** hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los ***threads*** o ***hilos*** de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un ***hilo*** de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método ***run()***, que es el que define la actividad principal de las ***threads***.

Los ***threads*** pueden ser ***daemon*** o ***no daemon***. Son ***daemon*** aquellos hilos que realizan en ***background*** (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un ***thread daemon*** podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de ***Java*** finaliza cuando sólo quedan corriendo ***threads*** de tipo ***daemon***. Por defecto, y si no se indica lo contrario, los ***threads*** son del tipo ***no daemon***.

6.1 CREACIÓN DE THREADS

En ***Java*** hay dos formas de crear nuevos ***threads***. La primera de ellas consiste en crear una nueva clase que herede de la clase ***java.lang.Thread*** y sobrecargar el método ***run()*** de dicha clase. El segundo método consiste en declarar una clase que implemente la interface ***java.lang.Runnable***, la cual declarará el método ***run()***; posteriormente se crea un objeto de tipo ***Thread*** pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface ***Runnable***). Como ya se ha apuntado, tanto la clase ***Thread*** como la interface ***Runnable*** pertenecen al package ***java.lang***, por lo que no es necesario importarlas.

A continuación se presentan dos ejemplos de creación de ***threads*** con cada uno de los dos métodos citados.

6.1.1 Creación de threads derivando de la clase Thread

Considérese el siguiente ejemplo de declaración de una nueva clase:

```
public class SimpleThread extends Thread {
    // constructor
    public SimpleThread (String str) {
        super(str);
    }

    // redefinición del método run()
    public void run() {
        for(int i=0;i<10;i++)
           System.out.println("Este es el thread : " + getName());
    }
}
```

En este caso, se ha creado la clase ***SimpleThread***, que hereda de ***Thread***. En su constructor se utiliza un ***String*** (opcional) para poner nombre al nuevo ***thread*** creado, y mediante ***super()*** se llama al constructor de la super-clase ***Thread***. Asimismo, se redefine el método ***run()***, que define la principal actividad del ***thread***, para que escriba 10 veces el nombre del ***thread*** creado.

Para poner en marcha este nuevo ***thread*** se debe crear un objeto de la clase ***SimpleThread***, y llamar al método ***start()***,heredado de la super-clase ***Thread***, que se encarga de llamar a ***run()***. Por ejemplo:

```
SimpleThread miThread = new SimpleThread(“Hilo de prueba”);
miThread.start();
```

6.1.2 Creación de threads implementando la interface Runnable

Esta segunda forma también requiere que se defina el método ***run()***, pero además es necesario crear un objeto de la clase ***Thread*** para lanzar la ejecución del nuevo hilo. Al constructor de la clase ***Thread*** hay que pasarle una referencia del objeto de la clase que implementa la interface ***Runnable***. Posteriormente, cuando se ejecute el método ***start()*** del ***thread***, éste llamará al método ***run()*** definido en la nueva clase. A continuación se muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface ***Runnable***:

```
public class SimpleRunnable implements Runnable {
    // se crea un nombre
    String nameThread;
    // constructor
    public SimpleRunnable (String str) {
        nameThread = str;
```

```
    }
    // definición del método run()
    public void run() {
        for(int i=0;i<10;i++)
           System.out.println("Este es el thread: " + nameThread);
    }
}
```

El siguiente código crea un nuevo ***thread*** y lo ejecuta por este segundo procedimiento:

```
SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
// se crea un objeto de la clase Thread pasándolo el objeto Runnable como argumento
Thread miThread = new Thread(p);
// se arranca el objeto de la clase Thread
miThread.start();
```

Este segundo método cobra especial interés con las ***applets***, ya que cualquier ***applet*** debe heredar de la clase ***java.applet.Applet***, y por lo tanto ya no puede heredar de ***Thread***. Véase el siguiente ejemplo:

```
class ThreadRunnable extends Applet implements Runnable {
    private Thread runner=null;
    // se redefine el método start() de Applet
    public void start() {
        if (runner == null) {
            runner = new Thread(this);
            runner.start(); // se llama al método start() de Thread
        }
    }
    // se redefine el método stop() de Applet
    public void stop(){
        runner = null; // se libera el objeto runner
    }
}
```

En este ejemplo, el argumento ***this*** del constructor de ***Thread*** hace referencia al objeto ***Runnable*** cuyo método ***run()*** debería ser llamado cuando el hilo ejecutado es un objeto de ***ThreadRunnable***.

La elección de una u otra forma -derivar de ***Thread*** o implementar ***Runnable***- depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un ***applet***, que siempre hereda de ***Applet***), no quedará más remedio que implementar ***Runnable***, aunque normalmente es más sencillo heredar de ***Thread***.

6.2 CICLO DE VIDA DE UN THREAD

En el apartado anterior se ha visto cómo crear nuevos objetos que permiten incorporar en un programa la posibilidad de realizar varias tareas simultáneamente. En la Figura 6.2 (tomada del ***Tutorial*** de **Sun**) se muestran los distintos estados por los que puede pasar un ***thread*** a lo largo de su vida. Un ***thread*** puede presentar cuatro estados distintos:

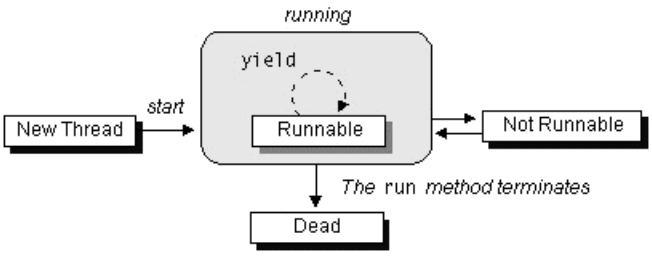


Figura 6.2. Ciclo de vida de un Thread.

1. *Nuevo* (***New***): El ***thread*** ha sido creado pero no inicializado, es decir, no se ha

ejecutado todavía el método ***start()***. Se producirá un mensaje de error (***IllegalThreadStateException***) si se intenta ejecutar cualquier método de la clase ***Thread*** distinto de ***start()***.

2. *Ejecutable* (***Runnable***): El ***thread*** puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro ***thread***.
3. *Bloqueado* (***Blocked*** o ***Not Runnable***): El ***thread*** podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un ***thread*** está en este estado, no se le asigna tiempo de CPU.
4. *Muerto* (***Dead***): La forma habitual de que un ***thread*** muera es finalizando el método ***run()***. También puede llamarse al método ***stop()*** de la clase ***Thread***, aunque dicho método es considerado "peligroso" y no se debe utilizar.

A continuación se explicarán con mayor detenimiento los puntos anteriores.

6.2.1 Ejecución de un nuevo thread

La creación de un nuevo ***thread*** no implica necesariamente que se empiece a ejecutar algo. Hace falta iniciarlo con el método ***start()***, ya que de otro modo, cuando se intenta ejecutar cualquier método del ***thread*** -distinto del método ***start()***- se obtiene en tiempo de ejecución el error ***IllegalThreadStateException***.

El método ***start()*** se encarga de llamar al método ***run()*** de la clase ***Thread***. Si el nuevo ***thread*** se ha creado heredando de la clase ***Thread*** la nueva clase deberá redefinirir el método ***run()*** heredado. En el caso de utilizar una clase que implemente la interface ***Runnable***, el método ***run()*** de la clase ***Thread*** se ocupa de llamar al método ***run()*** de la nueva clase (véase el Apartado 6.1.2, en la página 126).

Una vez que el método ***start()*** ha sido llamado, se puede decir ya que el ***thread*** está "corriendo" (***running***), lo cual no quiere decir que se esté ejecutando en todo momento, pues ese ***thread*** tiene que compartir el tiempo de la CPU con los demás ***threads*** que también estén ***running***. Por eso más bien se dice que dicha ***thread*** es ***runnable***.

6.2.2 Detener un Thread temporalmente: Runnable - Not Runnable

El sistema operativo se ocupa de asignar tiempos de CPU a los distintos ***threads*** que se estén ejecutando simultáneamente. Aun en el caso de disponer de un ordenador con más de un procesador (2 ó más CPUs), el número de ***threads*** simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el sistema continuamente asigna a los distintos ***threads*** en estado ***runnable*** se utilizan en ejecutar el método ***run()*** de cada ***thread***. Por diversos motivos, un ***thread*** puede en un determinado momento renunciar "voluntariamente" a su tiempo de CPU y otorgárselo al sistema para que se lo asigne a otro ***thread***. Esta "renuncia" se realiza mediante el método ***yield()***. Es importante que este método sea utilizado por las actividades que tienden a "monopolizar" la CPU. El método ***yield()*** viene a indicar que en ese momento no es muy importante para ese ***thread***

el ejecutarse continuamente y por lo tanto tener ocupada la CPU. En caso de que ningún ***thread*** esté requiriendo la CPU para una actividad muy intensiva, el sistema volverá casi de inmediato a asignar nuevo tiempo al ***thread*** que fue "generoso" con los demás. Por ejemplo, en un Pentium II 400 Mhz es posible llegar a más de medio millón de llamadas por segundo al método ***yield()***, dentro del método ***run()***, lo que significa que llamar al método ***yield()*** apenas detiene al ***thread***, sino que sólo ofrece el control de la CPU para que el sistema decida si hay alguna otra tarea que tenga mayor prioridad.

Si lo que se desea es parar o bloquear temporalmente un ***thread*** (pasar al estado ***Not Runnable***), existen varias formas de hacerlo:

1. Ejecutando el método ***sleep()*** de la clase ***Thread***. Esto detiene el ***thread*** un tiempo pre-establecido. De ordinario el método ***sleep()*** se llama desde el método ***run()***.
2. Ejecutando el método ***wait()*** heredado de la clase ***Object***, a la espera de que suceda algo que es necesario para poder continuar. El ***thread*** volverá nuevamente a la situación de ***runnable*** mediante los métodos ***notify()*** o ***notifyAll()***, que se deberán ejecutar cuando cesa la condición que tiene detenido al thread (ver Apartado 6.3, en la página 131).
3. Cuando el ***thread*** está esperando para realizar operaciones de Entrada/Salida o Input/Output (E/S ó I/O).
4. Cuando el ***thread*** está tratando de llamar a un método ***synchronized*** de un objeto, y dicho objeto está bloqueado por otro ***thread*** (véase el Apartado 6.3)

Un ***thread*** pasa automáticamente del estado ***Not Runnable*** a ***Runnable*** cuando cesa alguna de las condiciones anteriores o cuando se llama a ***notify()*** o ***notifyAll()***.

La clase ***Thread*** dispone también de un método ***stop()***, pero ***no se debe utilizar*** ya que puede provocar bloqueos del programa (***deadlock***). Hay una última posibilidad para detener un ***thread***, que consiste en ejecutar el método ***suspend()***. El ***thread*** volverá a ser ejecutable de nuevo ejecutando el método ***resume()***. Esta última forma también se desaconseja, por razones similares a la utilización del método ***stop()***.

El método ***sleep()*** de la clase ***Thread*** recibe como argumento el tiempo en *milisegundos* que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en *nanosegundos*. Las declaraciones de estos métodos son las siguientes:

```
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanosecons) throws InterruptedException
```

Considérese el siguiente ejemplo:

```
System.out.println ("Contador de segundos");
int count=0;
public void run () {
    try {
        sleep(1000);
        System.out.println(count++);
    } catch (InterruptedException e){}
}
```

Se observa que el método ***sleep()*** puede lanzar una ***InterruptedException*** que ha de ser capturada. Así se ha hecho en este ejemplo, aunque luego no se gestiona esa excepción.

La forma preferible de detener temporalmente un ***thread*** es la utilización conjunta de los métodos ***wait()*** y ***notifyAll()***. La principal ventaja del método ***wait()*** frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a ***wait()***:

1. Indicando el tiempo máximo que debe estar parado (en *milisegundos* y con la opción de indicar también *nanosegundos*), de forma análoga a ***sleep()***. A diferencia del método ***sleep()***, que simplemente detiene el ***thread*** el tiempo indicado, el método ***wait()*** establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos ***notify()*** o ***notifyAll()*** que indican la liberación de los objetos bloqueados, el ***thread*** continuará sin esperar a concluir el tiempo indicado. Las dos declaraciones del método ***wait()*** son como siguen:

   ```
   public final void wait(long timeout) throws InterruptedException
   public final void wait(long timeout, int nanos) throws InterruptedException
   ```

2. Sin argumentos, en cuyo caso el ***thread*** permanece parado hasta que sea reinicializado explícitamente mediante los métodos ***notify()*** o ***notifyAll()***.

   ```
   public final void wait() throws InterruptedException
   ```

Los métodos ***wait()*** y ***notify()*** han de estar incluidas en un método ***synchronized,*** ya que de otra forma se obtendrá una excepción del tipo ***IllegalMonitorStateException*** en tiempo de ejecución. El uso típico de ***wait()*** es el de esperar a que se cumpla alguna determinada condición, ajena al propio ***thread***. Cuando ésta se cumpla, se utilizará el método ***notifyAll()*** para avisar a los distintos ***threads*** que pueden utilizar el objeto. Estos nuevos conceptos se explican con más profundidad en el Apartado 6.3.

6.2.3 Finalizar un Thread

Un ***thread*** finaliza cuando el método ***run()*** devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un bucle ***for*** que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un bucle ***while*** en el método ***run()***). Es habitual poner las siguientes sentencias en el caso de ***Applets Runnables***:

```
public class MyApplet extends Applet implements Runnable {
    // se crea una referencia tipo Thread
    private Thread AppletThread;
    ...
    // método start() del Applet
    public void start() {
        if(AppletThread == null){          // si no tiene un objeto Thread asociado
           AppletThread = new Thread(this, "El propio Applet");
           AppletThread.start();           // se arranca el thread y llama a run()
        }
    }
    // método stop() del Applet
    public void stop() {
        AppletThread = null;               // iguala la referencia a null
    }
    // método run() por implementar Runnable
    public void run() {
        Thread myThread = Thread.currentThread();
        while (myThread == AppletThread) { // hasta que se ejecute stop() de Thread
           ...                             // código a ejecutar
        }
    }
} // fin de la clase MyApplet
```

donde ***AppletThread*** es el ***thread*** que ejecuta el método ***run()*** MyApplet. Para finalizar el thread basta poner la referencia ***AppletThread*** a ***null***. Esto se consigue en el ejemplo con el método ***stop()*** del ***applet*** (distinto del método ***stop()*** de la clase ***Thread***, que no conviene utilizar).

Para saber si un ***thread*** está "vivo" o no, es útil el método ***isAlive()*** de la clase ***Thread***, que devuelve ***true*** si el ***thread*** ha sido inicializado y no parado, y ***false*** si el ***thread*** es todavía nuevo (no ha sido inicializado) o ha finalizado.

6.3 SINCRONIZACIÓN

La ***sincronización*** nace de la necesidad de evitar que dos o más ***threads*** traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un ***thread*** tratara de escribir en un fichero, y otro ***thread*** estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada. Otra situación en la que hay que sincronizar ***threads*** se produce cuando un ***thread*** debe esperar a que estén preparados los datos que le debe suministrar el otro ***thread***. Para solucionar estos tipos de problemas es importante poder ***sincronizar*** los distintos ***threads***.

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos ***threads*** distintos se denominan ***secciones críticas*** (***critical sections***). Para sincronizar dos o más ***threads***, hay que utilizar el modificador ***synchronized*** en aquellos métodos del ***objeto-recurso*** con los que puedan producirse situaciones conflictivas. De esta forma, ***Java*** bloquea (asocia un ***bloqueo*** o ***lock***) con el recurso sincronizado. Por ejemplo:

```
public synchronized void metodoSincronizado() {
   ...// accediendo por ejemplo a las variables de un objeto
   ...
}
```

La ***sincronización*** previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como ***private*** (o ***protected***). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar ***sincronizados***.

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto pero otros no, el programa puede no funcionar correctamente. La razón es que los métodos no sincronizados pueden acceder libremente a las variables miembro, ignorando el bloqueo del objeto. Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados ***synchronized***. De esta forma, si algún método accede a un determinado recurso, ***Java*** bloquea dicho recurso, de forma que el resto de ***threads*** no puedan acceder al mismo hasta que el primero en acceder termine de realizar su tarea. ***Bloquear un recurso u objeto*** significa que sobre ese objeto no pueden actuar simultáneamente dos ***métodos sincronizados***.

Existen dos niveles de bloqueo de un recurso. El primero es ***a nivel de objetos***, mientras que el segundo es ***a nivel de clases***. El primero se consigue declarando todos los métodos de una clase como ***synchronized***. Cuando se ejecuta un método ***synchronized*** sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que si otro ***thread*** intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto). Si existen varios objetos de una misma clase, como

los bloqueos se producen a nivel de objeto, es posible tener distintos ***threads*** ejecutando métodos sobre diversos objetos de una misma clase.

El bloqueo de recursos ***a nivel de clases*** se corresponde con los ***métodos de clase*** o ***static***, y por lo tanto con las ***variables de clase*** o ***static***. Si lo que se desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como ***synchronized static***. Durante la ejecución de un método declarado de esta segunda forma ningún método sincronizado tendrá acceso a ningún objeto de la clase bloqueada.

La sincronización puede ser problemática y generar errores. Un ***thread*** podría bloquear un determinado recurso de forma indefinida, impidiendo que el resto de ***threads*** accedieran al mismo. Para evitar esto último, habrá que utilizar la sincronización sólo donde sea estrictamente necesario.

Es necesario tener presente que si dentro un método sincronizado se utiliza el método ***sleep()*** de la clase ***Thread***, el objeto bloqueado permanecerá en ese estado durante el tiempo indicado en el argumento de dicho método. Esto implica que otros ***threads*** no podrán acceder a ese objeto durante ese tiempo, aunque en realidad no exista peligro de simultaneidad ya que durante ese tiempo el thread que mantiene bloqueado el objeto no realizará cambios. Para evitarlo es conveniente sustituir ***sleep()*** por el método ***wait()*** de la clase ***java.lang.Object*** heredado automáticamente por todas las clases. Cuando se llama al método ***wait()*** (siempre debe hacerse desde un método o bloque ***synchronized***) se libera el bloqueo del objeto y por lo tanto es posible continuar utilizando ese objeto a través de métodos sincronizados. El método ***wait()*** detiene el ***thread*** hasta que se llame al método ***notify()*** o ***notifyAll()*** del objeto, o finalice el tiempo indicado como argumento del método ***wait()***. El método ***unObjeto.notify()*** lanza una señal indicando al sistema que puede activar uno de los ***threads*** que se encuentren bloqueados esperando para acceder al objeto ***unObjeto***. El método ***notifyAll()*** lanza una señal a todos los ***threads*** que están esperando la liberación del objeto.

Los métodos ***notify()*** y ***notifyAll()*** deben ser llamados desde el ***thread*** que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un ***thread*** se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto. Los bloqueos de tipo clase, se consiguen ejecutando un ***método de clase sincronizado*** (***synchronized static***). Véanse las dos funciones siguientes, de las que ***put()*** inserta un dato y ***get()*** lo recoge:

```
public synchronized int get() {
    while (available == false) {
        try {
            // Espera a que put() asigne el valor y lo comunique con notify()
            wait();
        } catch (InterruptedException e) { }
    }
    available = false;
    // notifica que el valor ha sido leído
    notifyAll();
    // devuelve el valor
    return contents;
}
public synchronized void put(int value) {
    while (available == true) {
        try {
            // Espera a que get() lea el valor disponible antes de darle otro
            wait();
        } catch (InterruptedException e) { }
    }
    // ofrece un nuevo valor y lo declara disponible
```

```
        contents = value;
        available = true;
        // notifica que el valor ha sido cambiado
        notifyAll();
    }
```

El bucle ***while*** de la función ***get()*** continúa ejecutándose (`avalaible == false`) hasta que el método ***put()*** haya suministrado un nuevo valor y lo indique con `avalaible = true`. En cada iteración del ***while*** la función ***wait()*** hace que el hilo que ejecuta el método ***get()*** se detenga hasta que se produzca un mensaje de que algo ha sido cambiado (en este caso con el método ***notifAll()*** ejecutado por ***put()***). El método ***put()*** funciona de forma similar.

Existe también la posibilidad de sincronizar una parte del código de un método sin necesidad de mantener bloqueado el objeto desde el comienzo hasta el final del método. Para ello se utiliza la palabra clave ***syncronized*** indicando entre paréntesis el objeto que se desea sincronizar (`synchronized(objetoASincronizar)`). Por ejemplo si se desea sincronizar el propio ***thread*** en una parte del método ***run()***, el código podría ser:

```
public void run() {
    while(true) {
        ...
        syncronized(this) { // El objeto a sincronizar es el propio thread
            ...             // Código sincronizado
        }
        try {
            sleep(500); // Se detiene el thread durante 0.5 segundos pero el objeto
                        // es accesible por otros threads al no estar sincronizado
        } catch(InterruptedException e) {}
    }
}
```

Un ***thread*** puede llamar a un método sincronizado de un objeto para el cual ya posee el bloqueo, volviendo a adquirir el bloqueo. Por ejemplo:

```
public class VolverAAdquirir {
    public synchronized void a() {
        b();
        System.out.println("Estoy en a()");
    }
    public synchronized void b() {
        System.out.println("Estoy en b()");
    }
}
```

El anterior ejemplo obtendrá como resultado:

```
Estoy en b()
Estoy en a()
```

debido a que se ha podido acceder al objeto con el método ***b()*** al ser el ***thread*** que ejecuta el método ***a()*** "propietario" con anterioridad del bloqueo del objeto.

La sincronización es un proceso que lleva bastante tiempo a la CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

6.4 PRIORIDADES

Con el fin de conseguir una correcta ejecución de un programa se establecen ***prioridades*** en los ***threads***, de forma que se produzca un reparto más eficiente de los recursos disponibles. Así, en un

determinado momento, interesará que un determinado proceso acabe lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos. La forma de llevar a cabo esto es gracias a las prioridades.

Cuando se crea un nuevo ***thread***, éste hereda la prioridad del ***thread*** desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase ***Thread***, que toman valores enteros que oscilan entre la máxima prioridad ***MAX_PRIORITY*** (normalmente tiene el valor 10) y la mínima prioridad ***MIN_PRIORITY*** (valor 1), siendo la prioridad por defecto ***NORM_PRIORITY*** (valor 5). Para modificar la prioridad de un ***thread*** se utiliza el método ***setPriority()***. Se obtiene su valor con ***getPriority()***.

El algoritmo de distribución de recursos en ***Java*** escoge por norma general aquel ***thread*** que tiene una prioridad mayor, aunque no siempre ocurra así, para evitar que algunos procesos queden "dormidos". Cuando hay dos o más ***threads*** de la misma prioridad (y además, dicha prioridad es la más elevada), el sistema no establecerá prioridades entre los mismos, y los ejecutará alternativamente dependiendo del sistema operativo en el que esté siendo ejecutado. Si dicho SO soporta el "time-slicing" (reparto del tiempo de CPU), como por ejemplo lo hace Windows 95/98/NT, los ***threads*** serán ejecutados alternativamente.

Un ***thread*** puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro ***thread*** de la misma prioridad, mediante el método ***yield()***, aunque en ningún caso a un ***thread*** de prioridad inferior.

6.5 GRUPOS DE THREADS

Todo hilo de ***Java*** debe formar parte de un grupo de hilos (***ThreadGroup***). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de ***threads*** proporcionan una forma sencilla de manejar múltiples ***threads*** como un solo objeto. Así, por ejemplo es posible parar varios ***threads*** con una sola llamada al método correspondiente. Una vez que un ***thread*** ha sido asociado a un ***threadgroup***, no puede cambiar de grupo.

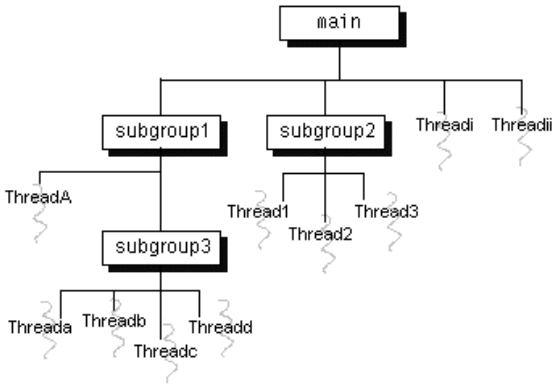


Figura 6.3. Representación de los grupos de Threads.

Cuando se arranca un programa, el sistema crea un ***ThreadGroup*** llamado ***main***. Si en la creación de un nuevo ***thread*** no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al ***threadgroup*** del ***thread*** desde el que ha sido creado (conocido como ***current thread group*** y ***current thread***, respectivamente). Si en dicho programa no se crea ningún ***ThreadGroup*** adicional, todos los ***threads*** creados pertenecerán al grupo ***main*** (en este grupo se encuentra el método ***main()***). La Figura 6.3 presenta una posible distribución de ***threads*** distribuidos en grupos de ***threads***.

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo ***thread***, según uno de los siguientes constructores:

```
public Thread (ThreadGroup grupo, Runnable destino)
```

```
public Thread (ThreadGroup grupo, String nombre)
public Thread (ThreadGroup grupo, Runnable destino, String nombre)
```

A su vez, un ***ThreadGroup*** debe pertenecer a otro ***ThreadGroup***. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ***ThreadGroup*** desde el que ha sido creado (por defecto al grupo ***main***). La clase ***ThreadGroup*** tiene dos posibles constructores:

```
ThreadGroup(ThreadGroup parent, String nombre);
ThreadGroup(String name);
```

el segundo de los cuales toma como ***parent*** el ***threadgroup*** al cual pertenezca el ***thread*** desde el que se crea (***Thread.currentThread()***). Para más información acerca de estos constructores, dirigirse a la documentación del API de ***Java*** donde aparecen numerosos métodos para trabajar con ***grupos de threads*** a disposición del usuario (***getMaxPriority()***, ***setMaxPriority()***, ***getName()***, ***getParent()***, ***parentOf()***).

En la práctica los ***ThreadGroups*** no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miThreadGroup, ”un thread para mi grupo");
```

donde se crea un grupo de ***threads*** (***miThreadGroup***) y un ***thread*** que pertenece a dicho ***grupo*** (***miThread)***.

7. APPLETS

7.1 QUÉ ES UN APPLET

Un ***applet*** es una mini-aplicación, escrita en ***Java***, que se ejecuta en un browser (***Netscape Navigator***, ***Microsoft Internet Explorer***, …) al cargar una página HTML que incluye información sobre el ***applet*** a ejecutar por medio de las ***tags*** <APPLET>... </APPLET>.

A continuación se detallan algunas características de las ***applets***:

1. Los ficheros de ***Java*** compilados (*.class) se descargan a través de la red desde un servidor de ***Web*** o servidor ***HTTP*** hasta el browser en cuya *Java Virtual Machine* se ejecutan. Pueden incluir también ficheros de imágenes y sonido.
2. Las ***applets*** no tienen ventana propia: se ejecutan en la ventana del browser (en un "***panel***").
3. Por la propia naturaleza "abierta" de Internet, las ***applets*** tienen importantes restricciones de seguridad, que se comprueban al llegar al browser: sólo pueden leer y escribir ficheros en el servidor del que han venido, sólo pueden acceder a una limitada información sobre el ordenador en el que se están ejecutando, etc. Con ciertas condiciones, las ***applets*** "de confianza" (***trusted applets***) pueden pasar por encima de estas restricciones.

Aunque su entorno de ejecución es un browser, las ***applets*** se pueden probar sin necesidad de browser con la aplicación ***appletviewer*** del JDK de ***Sun***.

7.1.1 Algunas características de las applets

Las características de las ***applets*** se pueden considerar desde el punto de vista del programador y desde el del usuario. En este manual lo más importante es el punto de vista del programador:

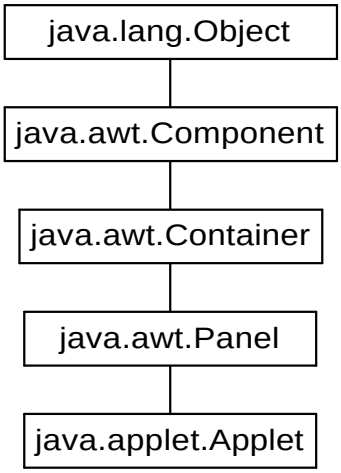


Figura 7.1. Jerarquía de clases de Applet.

- Las ***applets*** no tienen un método ***main()*** con el que comience la ejecución. El papel central de su ejecución lo asumen otros métodos que se verán posteriormente.
- Todas las ***applets*** derivan de la clase ***java.applet.Applet***. La Figura 7.1 muestra la jerarquía de clases de la que deriva la clase ***Applet***. Las ***applets*** deben redefinir ciertos métodos heredados de ***Applet*** que controlan su ejecución: ***init()***, ***start()***, ***stop()***, ***destroy()***.
- Se heredan otros muchos métodos de las super-clases de ***Applet*** que tienen que ver con la generación de interfaces gráficas de usuario (AWT). Así, los métodos gráficos se heredan de ***Component***, mientras que la capacidad de añadir componentes de interface de usuario se hereda de ***Container*** y de ***Panel***.
- Las ***applets*** también suelen redefinir ciertos métodos gráficos: los más importantes son ***paint()*** y ***update()***, heredados de ***Component*** y de ***Container***; y ***repaint()*** heredado de ***Component***.

- Las ***applets*** disponen de métodos relacionados con la obtención de información, como por ejemplo: ***getAppletInfo()***, ***getAppletContext()***, ***getParameterInfo()***, ***getParameter()***, ***getCodeBase()***, ***getDocumentBase()***, e ***isActive()***.

El método ***showStatus()*** se utiliza para mostrar información en la barra de estado del browser. Existen otros métodos relacionados con imágenes y sonido: ***getImage()***, ***getAudioClip()***, ***play()***, etc.

7.1.2 Métodos que controlan la ejecución de un applet

Los métodos que se estudian en este Apartado controlan la ejecución de las ***applets***. De ordinario el programador tiene que redefinir uno o más de estos métodos, pero no tiene que preocuparse de llamarlos: el browser se encarga de hacerlo.

7.1.2.1 Método init()

Se llama automáticamente al método ***init()*** en cuanto el browser o visualizador carga el ***applet***. Este método se ocupa de todas las tareas de inicialización, realizando las funciones del ***constructor*** (al que el browser no llama).

En ***Netscape Navigator*** se puede reinicializar un ***applet*** con ***Shift+Reload***.

7.1.2.2 Método start()

El método ***start()*** se llama automáticamente en cuanto el ***applet*** se hace visible, después de haber sido inicializada. Se llama también cada vez que el ***applet*** se hace de nuevo visible después de haber estado oculta (por dejar de estar activa esa página del browser, al cambiar el tamaño de la ventana del browser, al hacer ***reload***, etc.).

Es habitual crear ***threads*** en este método para aquellas tareas que, por el tiempo que requieren, dejarían sin recursos al ***applet*** o incluso al browser. Las animaciones y ciertas tareas a través de Internet son ejemplos de este tipo de tareas.

7.1.2.3 Método stop()

El método ***stop()*** se llama de forma automática al ocultar el ***applet*** (por haber haber dejado de estar activa la página del browser, por hacer ***reload*** o ***resize***, etc.).

Con objeto de no consumir recursos inútilmente, en este método se suelen parar las ***threads*** que estén corriendo en el ***applet***, por ejemplo para mostrar animaciones.

7.1.2.4 Método destroy()

Se llama a este método cuando el ***applet*** va a ser descargada para liberar los recursos que tenga reservados (excepto la memoria). De ordinario no es necesario redefinir este método, pues el que se hereda cumple bien con esta misión.

7.1.3 Métodos para dibujar el applet

Las ***applets*** son aplicaciones gráficas que aparecen en una zona de la ventana del browser. Por ello deben redefinir los métodos gráficos ***paint()*** y ***update()***. El método ***paint()*** se declara en la forma:

```
public void paint(Graphics g)
```

El objeto gráfico ***g*** pertenece a la clase ***java.awt.Graphics***, que siempre debe ser importada por el ***applet***. Este objeto define un contexto o estado gráfico para dibujar (métodos gráficos, colores, fonts, etc.) y es creado por el browser.

Todo el trabajo gráfico del ***applet*** (dibujo de líneas, formas gráficas, texto, etc.) se debe incluir en el método ***paint()***, porque este método es llamado cuando el ***applet*** se dibuja por primera vez y también de forma automática cada vez que el ***applet*** se debe redibujar.

En general, el programador crea el método ***paint()*** pero no lo suele llamar. Para pedir explícitamente al sistema que vuelva a dibujar el ***applet*** (por ejemplo, por haber realizado algún cambio) se utiliza el método ***repaint()***, que es más fácil de usar, pues no requiere argumentos. El método ***repaint()*** se encarga de llamar a ***paint()*** a través de ***update()***.

El método ***repaint()*** llama a ***update()***, que borra todo pintando de nuevo con el color de fondo y luego llama a ***paint()***. A veces esto produce parpadeo de pantalla o ***flickering***. Existen dos formas de evitar el ***flickering***:

1. Redefinir ***update()*** de forma que no borre toda la ventana sino sólo lo necesario.
2. Redefinir ***paint()*** y ***update()*** para utilizar ***doble buffer***.

Ambas formas fueron consideradas en los Apartados 5.6.1 y 5.6.2, en la página 123.

7.2 CÓMO INCLUIR UN APPLET EN UNA PÁGINA HTML

Para llamar a un ***applet*** desde una página HTML se utiliza la tag doble <APPLET>…</APPLET>, cuya forma general es (los elementos opcionales aparecen entre corchetes[]):

```
<APPLET CODE="miApplet.class" [CODEBASE="unURL"] [NAME="unName"]
    WIDTH="wpixels" HEIGHT="hpixels"
    [ALT="TextoAlternativo"]>
    [texto alternativo para browsers que reconocen el tag <applet> pero no pueden
           ejecutar el applet]
    [<PARAM NAME="MyName1" VALUE="valueOfMyName1">]
    [<PARAM NAME="MyName2" VALUE="valueOfMyName2">]
</APPLET>
```

El atributo NAME permite dar un nombre opcional al ***applet***, con objeto de poder comunicarse con otras ***applets*** o con otros elementos que se estén ejecutando en la misma página. El atributo ARCHIVE permite indicar uno o varios ficheros Jar o Zip (separados por comas) donde se deben buscar las clases.

A continuación se señalan otros posibles atributos de <APPLET>:

- ARCHIVE="file1, file2, file3". Se utiliza para especificar ficheros JAR y ZIP.
- ALIGN, VSPACE, HSPACE. Tienen el mismo significado que el tag IMG de HTML.

7.3 PASO DE PARÁMETROS A UN APPLET

Los tags PARAM permiten pasar diversos parámetros desde el fichero HTML al programa ***Java*** del ***applet***, de una forma análoga a la que se utiliza para pasar argumentos a ***main()***.

Cada parámetro tiene un ***nombre*** y un ***valor***. Ambos se dan en forma de ***String***, aunque el valor sea numérico. El ***applet*** recupera estos parámetros y, si es necesario, convierte los ***Strings*** en valores numéricos. El valor de los parámetros se obtienen con el siguiente método de la clase ***Applet***:

```
String getParameter(String name)
```

La conversión de ***Strings*** a los tipos primitivos se puede hacer con los métodos asociados a los ***wrappers*** que ***Java*** proporciona para dichos tipo fundamentales (Integer.parseInt(String), Double.valueOf(String), …). Estas clases de wrappers se estudiaron en el Apartado 4.3, a partir de la página 69.

En los ***nombres*** de los parámetros no se distingue entre mayúsculas y minúsculas, pero sí en los ***valores***, ya que serán interpretados por un programa ***Java***, que sí distingue.

El programador del ***applet*** debería prever siempre unos ***valores por defecto*** para los parámetros del ***applet***, para el caso de que en la página HTML que llama al ***applet*** no se definan.

El método ***getParameterInfo()*** devuelve una matriz de ***Strings*** (*String[][]*) con información sobre cada uno de los parámetros soportados por el ***applet***: *nombre*, *tipo* y *descripción*, cada uno de ellos en un ***String***. Este método debe ser redefinido por el programador del ***applet*** y utilizado por la persona que prepara la página HTML que llama al ***applet***. En muchas ocasiones serán personas distintas, y ésta es una forma de que el programador del ***applet*** dé información al usuario.

7.4 CARGA DE APPLETS

7.4.1 Localización de ficheros

Por defecto se supone que los ficheros ****.class*** del ***applet*** están en el mismo directorio que el fichero HTML. Si el ***applet*** pertenece a un ***package***, el browser utiliza el nombre del ***package*** para construir un ***path de directorio*** relativo al directorio donde está el HTML.

El atributo CODEBASE permite definir un URL para los ficheros que continen el código y demás elementos del ***applet***. Si el directorio definido por el URL de CODEBASE es *relativo*, se interpreta respecto al directorio donde está el HTML; si es *absoluto* se interpreta en sentido estricto y puede ser cualquier directorio de la Internet.

7.4.2 Archivos JAR (Java Archives)

Si un ***applet*** consta de varias clases, cada fichero ****.class*** requiere una conexión con el servidor de Web (servidor de protocolo HTTP), lo cual puede requerir algunos segundos. En este caso es conveniente agrupar todos los ficheros en un archivo único, que se puede comprimir y cargar con una sola conexión HTTP.

Los archivos JAR están basados en los archivos ZIP y pueden crearse con el programa ***jar*** que viene con el JDK. Por ejemplo:

```
jar cvf myFile.jar *.class *.gif
```

crea un fichero llamado ***myFile.jar*** que contiene todos los ficheros ****.class*** y ****.gif*** del directorio actual. Si las clases pertenecieran a un package llamado ***es.ceit.infor2*** se utilizaría el comando:

```
jar cvf myFile.jar es\ceit\infor2\*.class *.gif
```

7.5 COMUNICACIÓN DEL APPLET CON EL BROWSER

La comunicación entre el applet y el browser en el que se está ejecutando se puede controlar mediante la interface ***AppletContext*** (package ***java.applet***). ***AppletContext*** es una interface implementada por el browser, cuyos métodos pueden ser utilizados por el ***applet*** para obtener información y realizar ciertas operaciones, como por ejemplo sacar ***mensajes breves*** en la ***barra de estado*** del browser. Hay que tener en cuenta que la barra de estado es compartida por el browser y las ***applets***, lo que tiene el peligro de que el mensaje sea rápidamente sobre-escrito por el browser u otras ***applets*** y que el usuario no llegue a enterarse del mensaje.

Los mensajes breves a la barra de estado se producen con el método ***showStatus()***, como por ejemplo,

```
getAppletContext().showStatus("Cargado desde el fichero " + filename);
```

Los mensajes más importantes se deben dirigir a la ***salida estándar*** o a la ***salida de errores***, que en ***Netscape Navigator*** es la ***Java Console*** (la cual se hace visible desde el menú ***Options*** en ***Navigator 3.0***, desde el menú ***Communicator*** en ***Navigator 4.0**** y desde ***Communicator/Tools*** en ***Navigator 4.5***). Estos mensajes se pueden enviar con las sentencias:

```
System.out.print();
System.out.println();
System.error.print();
System.error.println();
```

Para mostrar documentos HTML en una ventana del browser se pueden utilizar los métodos siguientes:

- ***showDocument(URL miUrl, [String target])***, que muestra un documento HTML en el *frame* del browser indicado por *target* (*name*, *_top*, *_parent*, *_blank*, *_self*).
- ***showDocument(URL miUrl)***, que muestra un documento HTML en la ventana actual del browser.

Un ***applet*** puede conseguir información de otras ***applets*** que están corriendo en la misma página del browser, enviarles mensajes y ejecutar sus métodos. El mensaje se envía invocando los métodos del otro ***applet*** con los argumentos apropiados.

Algunos browsers exigen, para que las ***applets*** se puedan comunicar, que las ***applets*** provengan del mismo browser o incluso del mismo directorio (que tengan el mismo *codebase*).

Por ejemplo, para obtener información de otras applets se pueden utilizar los métodos:

- ***getApplet(String name)***, que devuelve el ***applet*** llamada ***name*** (o ***null*** si no la encuentra). El nombre del ***applet*** se pone con el atributo opcional NAME o con el parámetro NAME.
- ***getApplets()***, que devuelve una *enumeración* con todas las ***applets*** de la página.

Para poder utilizar todos los métodos de un applet que se está ejecutando en la misma página HTML (y no sólo los métodos comunes heredados de ***Applet***), debe hacerse un ***cast*** del objeto de la clase ***Applet*** que se obtiene como valor de retorno de ***getApplet()*** a la clase concreta del ***applet***.

Para que pueda haber ***respuesta*** (es decir, comunicación en los dos sentidos), el primer applet que envía un mensaje debe enviar una referencia a sí misma por medio del argumento ***this***.

7.6 SONIDOS EN APPLETS

La clase ***Applet*** y la interface ***AudioClips*** permiten utilizar sonidos en applets. La Tabla 7.1 muestra

Métodos de Applet	**Función que realizan**
public AudioClip getAudioClip(URL url)	Devuelve el objeto especificado por url, que implementa la interface AudioClip
public AudioClip getAudioClip(URL url, String name)	Devuelve el objeto especificado por url (dirección base) y name (dirección relativa)
play(URL url), play(URL url, String name)	Hace que suene el AudioClip correspondiente a la dirección especificada
Métodos de la interface AudioClip (en java.applet)	**Función que realizan**
void loop()	Ejecuta el sonido repetidamente
void play()	Ejecuta el sonido una sola vez
void stop()	Detiene el sonido

Tabla 7.1. Métodos de Applet y de AudioClip relacionados con sonidos.

algunos métodos interesantes al respecto.

Respecto a la carga de sonidos, por lo general es mejor cargar los sonidos en un ***thread*** distinto (creado en el método ***init()***) que en el propio método ***init()***, que tardaría en devolver el control y permitir al usuario empezar a interaccionar con el ***applet***.

Si el sonido no ha terminado de cargarse (en la ***thread*** especial para ello) y el usuario interacciona con el ***applet*** para ejecutarlo, el ***applet*** puede darle un aviso de que no se ha terminado de cargar.

7.7 IMÁGENES EN APPLETS

Las ***applets*** admiten los formatos JPEG y GIF para representar imágenes a partir de ficheros localizados en el servidor. Estas imágenes se pueden cargar con el método ***getImage()*** de la clase ***Applet***, que puede tener las formas siguientes:

```
public Image getImage(URL url)
public Image getImage(URL url, String name)
```

Estos métodos devuelven el control inmediatamente. Las imágenes de cargan cuando se da la orden de dibujar las imágenes en la pantalla. El dibujo se realiza entonces de forma incremental, a medida que el contenido va llegando.

Para dibujar imágenes se utiliza el método ***drawImage()*** de la clase ***Graphics***, que tiene las formas siguientes:

```
public abstract boolean drawImage(Image img, int x, int y,
                                  Color bgcolor, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, int width, int height,
                                  Color bgcolor, ImageObserver observer)
```

El primero de ellos dibuja la imagen con su tamaño natural, mientras que el segundo realiza un cambio en la escala de la imagen.

Los métodos ***drawImage()*** van dibujando la parte de la imagen que ha llegado, con su tamaño, a partir de las coordenadas (x, y) indicadas, utilizando ***bgcolor*** para los pixels transparentes.

Estos métodos devuelven el control inmediatamente, aunque la imagen no esté del todo cargada. En este caso devuelve ***false***. En cuanto se carga una parte adicional de la imagen, el proceso que realiza el dibujo avisa al ***ImageObserver*** especificado. ***ImageObserver*** es una interface implementada por ***Applet*** que permite seguir el proceso de carga de una imagen.

7.8 OBTENCIÓN DE LAS PROPIEDADES DEL SISTEMA

Un ***applet*** puede obtener información del sistema o del entorno en el que se ejecuta. Sólo algunas propiedades del sistema son accesibles. Para acceder a las propiedades del sistema se utiliza un método ***static*** de la clase ***System***:

```
String salida = System.getProperty("file.separator");
```

Los nombres y significados de las propiedades del sistema accesibles son las siguientes:

"file.separator"	Separador de directorios (por ejemplo, "/" o "\")
"java.class.version"	Número de version de las clases de ***Java***
"java.vendor"	Nombre específico del vendedor de Java
"java.vendor.url"	URL del vendedor de Java
"java.version"	Número de versión Java
"line.separator"	Separador de líneas
"os.arch"	Arquitectura del sistema operativo
"os.name"	Nombre del sistema operativo
"path.separator"	Separador en la variable Path (por ejemplo, ":"

No se puede acceder a las siguientes propiedades del sistema: "java.class.path", "java.home", "user.dir", "user.home", "user.name".

7.9 UTILIZACIÓN DE THREADS EN APPLETS

Un ***applet*** puede ejecutarse con varias ***threads***, y en muchas ocasiones será necesario o conveniente hacerlo así. Hay que tener en cuenta que un ***applet*** se ejecuta siempre en un browser (o en la aplicación ***appletviewer***).

Así, las ***threads*** en las que se ejecutan los métodos mayores -***init()***, ***start()***, ***stop()*** y ***destroy()***- dependen del browser o del entorno de ejecución. Los métodos gráficos -***paint()***, ***update()*** y ***repaint()***- se ejecutan siempre desde una ***thread*** especial del AWT.

Algunos browsers dedican un ***thread*** para cada ***applet*** en una misma página; otros crean un grupo de ***threads*** para cada ***applet*** (para poderlas matar al mismo tiempo, por ejemplo). En cuelquier caso se garantiza que todas las ***threads*** creadas por los métodos mayores pertenecen al mismo grupo.

Se deben introducir ***threads*** en ***applets*** siempre que haya tareas que consuman mucho tiempo (cargar una imagen o un sonido, hacer una conexión a Internet, …). Si estas tareas pesadas se ponen

en el método ***init()*** bloquean cualquier actividad del ***applet*** o incluso de la página HTML hasta que se completan. Las tareas pesadas pueden ser de dos tipos:

- Las que sólo se hacen una vez.
- Las que se repiten muchas veces.

Un ejemplo de tarea que se repite muchas veces puede ser una animación. En este caso, la tarea repetitiva se pone dentro de un bucle ***while*** o ***do…while***, dentro del ***thread***. El ***thread*** se debería crear dentro del método ***start()*** del ***applet*** y destruirse en ***stop()***. De este modo, cuando el ***applet*** no está visible se dejan de consumir recursos.

Al crear el ***thread*** en el método ***start()*** se pasa una referencia al ***applet*** con la palabra ***this***, que se refiere al ***applet***. El applet deberá implementar la interface ***Runnable***, y por tanto debe definir el método ***run()***, que es el centro del ***Thread*** (ver Apartado 6.1.2, en la página 126).

Un ejemplo de tarea que se realiza una sola vez es la carga de imágenes ****.gif*** o ****.jpeg***, que ya se realiza automáticamente en un ***thread*** especial.

Sin embargo, los sonidos no se cargan en ***threads*** especiales de forma automática; los debe crear el programador para cargarlos en "background". Este es un caso típico de programa ***producer-consumer***: el ***thread*** es el *producer* y el ***applet*** el *consumer*. Las ***threads*** deben estar ***sincronizadas***, para lo que se utilizan los métodos ***wait()*** y ***notifyAll()***.

A continuación se presenta un ejemplo de ***thread*** con tarea repetitiva:

```
public void start() {
    if (repetitiveThread == null) {
        repetitiveThread = new Thread(this); // se crea un nuevo thread
    }
    repetitiveThread.start(); // se arranca el thread creado: start() llama a run()
}

public void stop() {
    repetitiveThread = null; // para parar la ejecución del thread
}


public void run() {
    ...
    while (Thread.currentThread() == repetitiveThread) {
        ... // realizar la tarea repetitiva.
    }
}
```

El método ***run()*** se detendrá en cuanto se ejecute el método ***stop()***, porque la referencia al ***thread*** está a ***null***.

7.10 APPLETS QUE TAMBIÉN SON APLICACIONES

Es muy interesante desarrollar ***aplicaciones*** que pueden funcionar también como ***applets*** y viceversa. En concreto, para hacer que un ***applet*** pueda ejecutarse como ***aplicación*** pueden seguirse las siguientes instrucciones:

1. Se añade un método ***main()*** a la clase ***MiApplet*** (que deriva de ***Applet***)

2. El método ***main()*** debe crear un objeto de la clase ***MiApplet*** e introducirlo en un ***Frame***.
3. El método ***main()*** debe también ocuparse de hacer lo que haría el browser, es decir, llamar a los métodos ***init()*** y ***start()*** de la clase ***MiApplet***.
4. Se puede añadir también una ***static inner class*** que derive de ***WindowAdapter*** y que gestione el evento de cerrar la ventana de la aplicación definiendo el método ***windowClosing()***. Este método llama al método ***System.exit(0)***. Según como sea el ***applet***, el método ***windowClosing()*** previamente deberá también llamar a los métodos ***MiApplet.stop()*** y ***MiApplet.destroy()***, cosa que para las ***applets*** se encarga de hacer el browser. En este caso conviene que el objeto de ***MiApplet*** creado por ***main()*** sea ***static***, en lugar de una variable local.

A continuación se presenta un ejemplo:

```
public class MiApplet extends Applet {
    ...
    public void init() {...}
    ...

    // clase para cerrar la aplicación
    static class WL extends WindowsAdapter {
        public void windowClosing(WindowEvent e) {
            MiApplet.stop();
            MiApplet.destroy();
            System.exit(0);
        }
    } // fin de WindowAdapter

    // programa principal
    public static void main(String[] args) {
        static MiApplet unApplet = new MiApplet();
        Frame unFrame = new Frame("MiApplet");
        unFrame.addWindowListener(new WL());
        unFrame.add(unapplet, BorderLayout.CENTER);
        unFrame.setSize(400,400);
        unApplet.init();
        unApplet.start();
        unFrame.setVisible(true);
    }

} // fin de la clase MiApplet
```

8. EXCEPCIONES

A diferencia de otros lenguajes de programación orientados a objetos como C/C++, ***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 durante este periodo. 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, 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 (indicando una nueva localización del fichero no encontrado).

Un buen programa debe gestionar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos "estilos" de hacer esto:

1. ***A la "antigua usanza"***: los métodos devuelven un código de error. Este código se chequea en el entorno que ha llamado al método con una serie de ***if elseif*** …, gestionando de forma diferente el resultado correcto o cada uno de los posibles errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.
2. ***Con soporte en el propio lenguaje***: En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores o ***Exceptions***. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y ***Java***.

En los siguientes apartados se examina cómo se trabaja con los bloques y expresiones ***try***, ***catch***, ***throw***, ***throws*** y ***finally***, cuándo se deben lanzar excepciones, cuándo se deben capturar y cómo se crean las clases propias de tipo ***Exception***.

8.1 EXCEPCIONES ESTÁNDAR DE JAVA

Los errores se representan mediante dos tipos de clases derivadas de la clase ***Throwable***: ***Error*** y ***Exception***. La siguiente figura muestra parcialmente la jerarquía de clases relacionada con ***Throwable***:

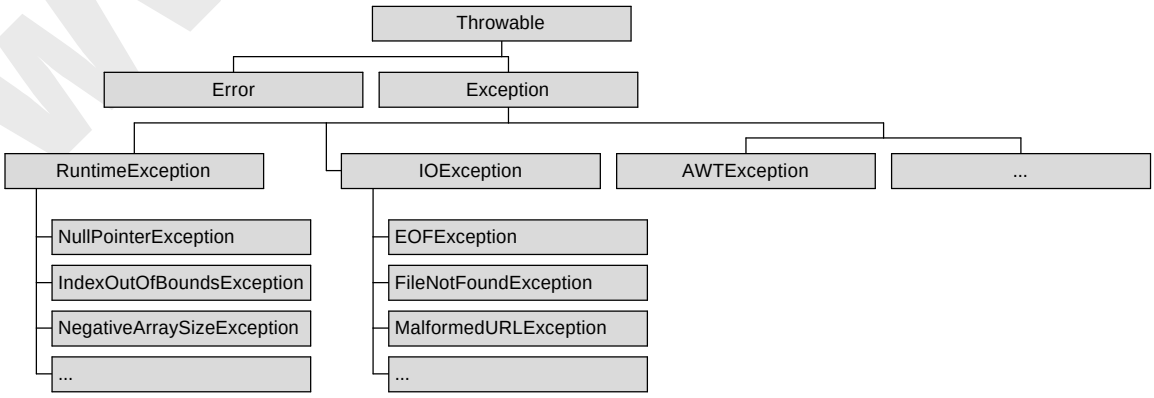


Figura 8.1: Jerarquía de clases derivadas de Throwable.

La clase ***Error*** está relacionada con errores de compilación, del sistema o de la JVM. De ordinario estos errores son ***irrecuperables*** y no dependen del programador ni debe preocuparse de capturarlos y tratarlos.

La clase ***Exception*** tiene más interés. Dentro de ella se puede distinguir:

1. ***RuntimeException***: Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar ***excepciones implícitas***.
2. Las demás clases derivadas de ***Exception*** son ***excepciones explícitas***. ***Java*** obliga a tenerlas en cuenta y chequear si se producen.

El caso de ***RuntimeException*** es un poco especial. El propio ***Java*** durante la ejecución de un programa chequea y lanza automáticamente las excepciones que derivan de ***RuntimeException***. El programador no necesita establecer los bloques ***try***/***catch*** para controlar este tipo de excepciones. Representan dos casos de errores de programación:

1. Un error que normalmente no suele ser chequeado por el programador, como por ejemplo recibir una referencia ***null*** en un método.
2. Un error que el programador debería haber chequeado al escribir el código, como sobrepasar el tamaño asignado de un array (genera un *ArrayIndexOutOfBoundsException* automáticamente).

En realidad sería posible comprobar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara chequear continuamente todo tipo de errores (que las ***referencias*** son distintas de ***null***, que todos los argumentos de los métodos son correctos, y un largo etcétera).

Las clases derivadas de ***Exception*** pueden pertenecer a distintos packages de ***Java***. Algunas perenecen a ***java.lang*** (*Throwable*, *Exception*, *RuntimeException*, …); otras a ***java.io*** (*EOFException*, *FileNotFoundException*, ...) o a otros packages. Por heredar de ***Throwable*** todos los tipos de excepciones pueden usar los métodos siguientes:

1. String ***getMessage()*** Extrae el mensaje asociado con la excepción.
2. String ***toString()*** Devuelve un String que describe la excepción.
3. void ***printStackTrace()*** Indica el método donde se lanzó la excepción.

8.2 LANZAR UNA EXCEPTION

Cuando en un método se produce una situación anómala es necesario lanzar una excepción. El proceso de lanzamiento de una excepción es el siguiente:

1. Se crea un objeto ***Exception*** de la clase adecuada.
2. Se lanza la excepción con la sentencia ***throw*** seguida del objeto ***Exception*** creado.

```
// Código que lanza la excepción MyException una vez detectado el error
MyException me = new MyException("MyException message");
throw me;
```

Esta excepción deberá ser capturada (***catch***) y gestionada en el propio método o en algún otro lugar del programa (en otro método anterior en la ***pila*** o ***stack*** de llamadas), según se explica en el Apartado 8.3.

Al lanzar una excepción el método termina de inmediato, sin devolver ningún valor. Solamente en el caso de que el método incluya los bloques ***try/catch/finally*** se ejecutará el bloque ***catch*** que la captura o el bloque ***finally*** (si existe).

Todo método en el que se puede producir uno o más tipos de excepciones (y que no utiliza directamente los bloques ***try/catch/finally*** para tratarlos) debe ***declararlas*** en el encabezamiento de la función por medio de la palabra ***throws***. Si un método puede lanzar varias excepciones, se ponen detrás de ***throws*** separadas por comas, como por ejemplo:

```
public void leerFichero(String fich) throws EOFException, FileNotFoundException {…}
```

Se puede poner únicamente una ***superclase de excepciones*** para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas. El caso anterior sería equivalente a:

```
public void leerFichero(String fich) throws IOException {…}
```

Las excepciones pueden ser lanzadas directamente por ***leerFichero()*** o por alguno de los métodos llamados por ***leerFichero()***, ya que las clases ***EOFException*** y ***FileNotFoundException*** derivan de ***IOException***.

Se recuerda que no hace falta avisar de que se pueden lanzar objetos de la clases ***Error*** o ***RuntimeException*** (excepciones implícitas).

8.3 CAPTURAR UNA EXCEPTION

Como ya se ha visto, ciertos métodos de los packages de ***Java*** y algunos métodos creados por cualquier programador producen ("lanzan") excepciones. Si el usuario llama a estos métodos sin tenerlo en cuenta se produce un error de compilación con un mensaje del tipo: "… *Exception java.io.IOException must be caugth or it must be declared in the throws clause of this method*". El programa no compilará mientras el usuario no haga una de estas dos cosas:

1. ***Gestionar la excepción*** con una construcción del tipo ***try {…} catch {…}***.
2. ***Re-lanzar la excepción*** hacia un método anterior en el ***stack***, declarando que su método también lanza dicha excepción, utilizando para ello la construcción ***throws*** en el header del método.

El compilador obliga a capturar las llamadas ***excepciones explícitas***, pero no protesta si se captura y luego no se hace nada con ella. En general, es conveniente por lo menos imprimir un mensaje indicando qué tipo de excepción se ha producido.

8.3.1 Bloques try y catch

En el caso de las excepciones que no pertenecen a las ***RuntimeException*** y que por lo tanto ***Java*** obliga a tenerlas en cuenta habrá que utilizar los bloques ***try***, ***catch*** y ***finally***. El código dentro del bloque ***try*** está "vigilado": Si se produce una situación anormal y se lanza por lo tanto una excepción el control salta o sale del bloque ***try*** y 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 sean necesarios, cada uno de los cuales tratará un tipo de excepción.

Las excepciones se pueden capturar individualmente o en grupo, por medio de una superclase de la que deriven todas ellas.

El bloque ***finally*** es opcional. Si se incluye sus sentencias se ejecutan siempre, sea cual sea la excepción que se produzca o si no se produce ninguna. El bloque ***finally*** se ejecuta aunque en el bloque ***try*** haya un ***return***.

En el siguiente ejemplo se presenta un método que debe "controlar" una ***IOException*** relacionada con la lectura ficheros y una ***MyException*** propia:

```
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
```

8.3.2 Relanzar una Exception

Existen algunos casos en los cuales el código de un método puede generar una ***Exception*** y no se desea incluir en dicho método la gestión del error. ***Java*** permite que este método pase o relance (***throws***) la ***Exception*** al método desde el que ha sido llamado, sin incluir en el método los bucles ***try/catch*** correspondientes. Esto se consigue mediante la adición de ***throws*** más el 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()***.

El ejemplo anterior (***metodo1***) realizaba la gestión de las excepciones dentro del propio método. Ahora se presenta un nuevo ejemplo (***metodo2***) que relanza las excepciones al siguiente método:

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

Según lo anterior, si un método llama a otros métodos que pueden lanzar excepciones (por ejemplo de un package de ***Java***), tiene 2 posibilidades:

1. ***Capturar*** las posibles excepciones y gestionarlas.
2. Desentenderse de las excepciones y ***remitirlas*** hacia otro método anterior en el ***stack*** para éste se encargue de gestionarlas.

Si no hace ninguna de las dos cosas anteriores el compilador da un error, salvo que se trate de una ***RuntimeException***.

8.3.3 Método finally {...}

El bloque ***finally {...}*** debe ir detrás de todos los bloques ***catch*** considerados. Si se incluye (ya que es opcional) sus sentencias se ejecutan siempre, sea cual sea el tipo de excepción que se produzca, o ***incluso si no se produce ninguna***. El bloque ***finally*** se ejecuta incluso si dentro de los bloques ***try/catch*** hay una sentencia ***continue***, ***break*** o ***return***. La forma general de una sección donde se controlan las excepciones es por lo tanto:

```
try {
    // Código "vigilado" que puede lanzar una excepción de tipo A, B o C
} catch (A a1) {
    // Se ocupa de la excepción A
} catch (B b1) {
    // Se ocupa de la excepción B
} catch (C c1) {
    // Se ocupa de la excepción C
} finally {
    // Sentencias que se ejecutarán en cualquier caso
}
```

El bloque ***finally*** es necesario en los casos en que se necesite recuperar o devolver a su situación original algunos elementos. No se trata de liberar la memoria reservada con ***new*** ya que de ello se ocupará automáticamente el *garbage collector*.

Como ejemplo se podría pensar en un bloque ***try*** dentro del cual se abre un fichero para lectura y escritura de datos y se desea cerrar el fichero abierto. El fichero abierto se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un fichero abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes a cerrar el fichero dentro del bloque ***finally***.

8.4 CREAR NUEVAS EXCEPCIONES

El programador puede crear sus propias excepciones sólo con heredar de la clase ***Exception*** o de una de sus clases derivadas. Lo lógico es heredar de la clase de la jerarquía de ***Java*** que mejor se adapte al tipo de excepción. Las clases ***Exception*** suelen tener dos constructores:

1. Un ***constructor*** sin argumentos.
2. Un ***constructor*** que recibe un ***String*** como argumento. En este ***String*** se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva ***super(String)***.

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

```
class MiExcepcion extends Exception {
    public MiExcepcion() {                  // Constructor por defecto
        super();
    }
    public MiExcepción(String s) {          // Constructor con mensaje
        super(s);
    }
}
```

8.5 HERENCIA DE CLASES Y TRATAMIENTO DE EXCEPCIONES

Si un método redefine otro método de una super-clase que utiliza ***throws***, el método de la clase derivada no tiene obligatoriamente que poder lanzar todas las mismas excepciones de la clase base. Es posible en el método de la subclase lanzar ***las mismas excepciones o menos***, pero no se pueden lanzar más excepciones. No puede tampoco lanzar nuevas excepciones ni excepciones de una clase más general.

Se trata de una restricción muy útil ya que como consecuencia de ello el código que funciona con la clase base podrá trabajar automáticamente con referencias de clases derivadas, incluyendo el tratamiento de excepciones, concepto fundamental en la *Programación Orientada a Objetos* (***polimorfismo***).

9. ENTRADA/SALIDA DE DATOS EN JAVA 1.1

Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar, como para devolver los resultados obtenidos.

La manera de representar estas entradas y salidas en ***Java*** es a base de ***streams*** (flujos de datos). Un ***stream*** es una conexión entre el programa y la fuente o destino de los datos. La información se traslada ***en serie*** (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un ***stream*** que conecta el monitor al programa. Se da a ese ***stream*** la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet o la lectura de la información de un sensor a través del puerto en serie.

9.1 CLASES DE JAVA PARA LECTURA Y ESCRITURA DE DATOS

El package ***java.io*** contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este package existen dos familias de jerarquías distintas para la entrada/salida de datos. La diferencia principal consiste en que una opera con ***bytes*** y la otra con ***caracteres*** (el carácter de ***Java*** está formado por dos bytes porque sigue el código ***Unicode***). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.

Desde ***Java 1.0***, la entrada y salida de datos del programa se podía hacer con clases derivadas de ***InputStream*** (para lectura) y ***OutputStream*** (para escritura). Estas clases tienen los métodos básicos ***read()*** y ***write()*** que manejan ***bytes*** y que no se suelen utilizar directamente. La Figura 9.1 muestra las clases que derivan de ***InputStream*** y la Figura 9.2 las que derivan de ***OutputStream***.

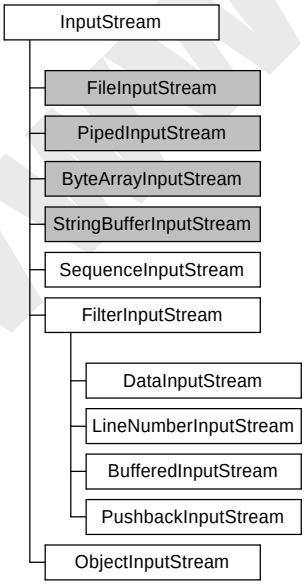


Figura 9.1. Jerarquía de clases InputStream.

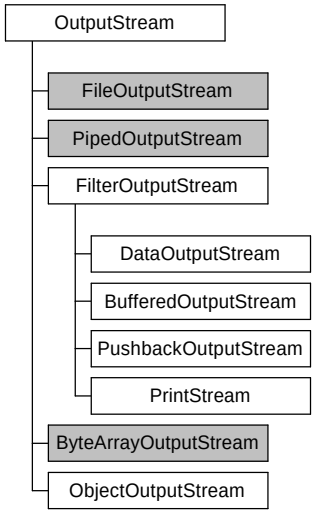


Figura 9.2. Jerarquía de clases OutputStream.

En ***Java 1.1*** aparecieron dos nuevas familias de clases, derivadas de ***Reader*** y ***Writer***, que manejan ***caracteres*** en vez de ***bytes***. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto. Las clases que heredan de ***Reader*** están incluidas en la Figura 9.3 y las que heredan de ***Writer*** en la Figura 9.4.

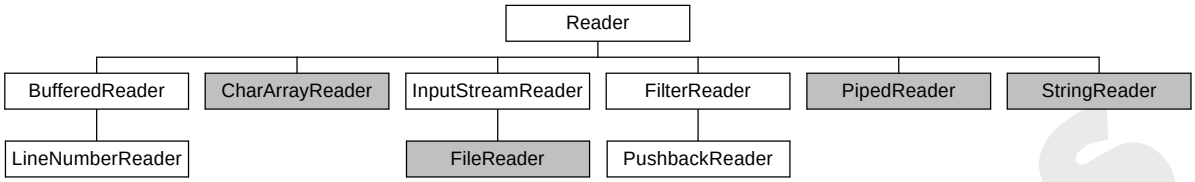


Figura 9.3. Jerarquía de clases Reader.

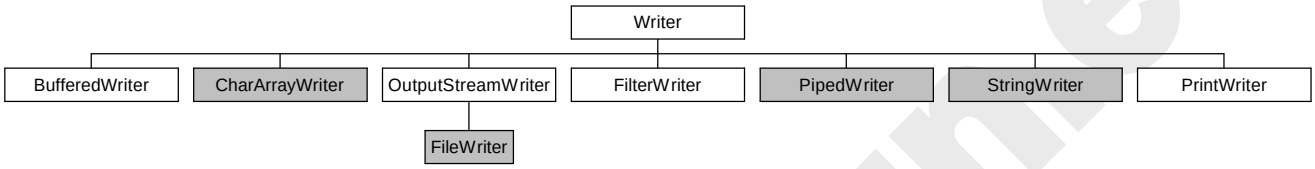


Figura 9.4. Jerarquía de clases Writer.

En las cuatro últimas figuras las clases con ***fondo gris*** definen de dónde o a dónde se están enviando los datos, es decir, el dispositivo con que conecta el ***stream***. Las demás (***fondo blanco***) añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new FileReader("autoexec.bat"));
```

Con esta línea se ha creado un ***stream*** que permite leer del archivo ***autoexec.bat***. Además, se ha creado a partir de él un objeto ***BufferedReader*** (que aporta la característica de utilizar ***buffer***[6]). Los caracteres que lleguen a través del ***FileReader*** pasarán a través del ***BufferedReader***, es decir utilizarán el ***buffer***.

A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación (***clases en gris***) y luego se le añadiran otras características (***clases en blanco***).

Se recomienda utilizar siempre que sea posible las clases ***Reader*** y ***Writer***, dejando las de ***Java 1.0*** para cuando sean imprescindibles. Algunas tareas como la ***serialización*** y la ***compresión*** necesitan las clases ***InputStream*** y ***OutputStream***.

9.1.1 Los nombres de las clases de java.io

Las clases de ***java.io*** siguen una nomenclatura sistemática que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la Tabla 9.1.

[6] Un ***buffer*** es un espacio de memoria intermedia que actúa de "colchón" de datos. Cuando se necesita un dato del disco se trae a memoria ese dato y sus datos contiguos, de modo que la siguiente vez que se necesite algo del disco la probabilidad de que esté ya en memoria sea muy alta. Algo similar se hace para escritura, intentando realizar en una sola operación de escritura física varias sentencias individuales de escritura.

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

Tabla 9.1. Palabras identificativas de las clases de java.io.

9.1.2 Clases que indican el origen o destino de los datos

La Tabla 9.2 explica el uso de las clases que definen el lugar con que conecta el ***stream***.

Clases	Función que realizan
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en **archivos** de disco. Se explicarán luego con más detalle.
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la **memoria** del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un "tubo" o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. Tambíen puede comunicar a dos programas distintos.

Tabla 9.2. Clases que indican el origen o destino de los datos.

9.1.3 Clases que añaden características

La Tabla 9.3 explica las funciones de las clases que alteran el comportamiento de un ***stream*** ya definido.

Clases	Función que realizan
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un **buffer** al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. ***BufferedReader*** por ejemplo tiene el método ***readLine()*** que lee una línea y la devuelve como un String.
InputStreamReader, OutputStreamWriter	Son clases puente que permiten **convertir** streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa.
ObjectInputStream, ObjectOutputStream	Pertenecen al mecanismo de la **serialización** y se explicarán más adelante.
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos **filtros** o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida.
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible , pero independiente de plataforma y se usan por tanto para **almacenaje** o para **transmisiones** entre ordenadores de distinto funcionamiento.
PrintWriter, PrintStream	Tienen métodos adaptados para **imprimir** las variables de Java con la apariencia normal. A partir de un boolean escriben “true” o “false”, colocan la coma de un número decimal, etc.

Tabla 9.3. Clases que añaden características.

9.2 ENTRADA Y SALIDA ESTÁNDAR (TECLADO Y PANTALLA)

En ***Java***, la entrada desde teclado y la salida a pantalla están reguladas a traves de la clase ***System***. Esta clase pertenece al package ***java.lang*** y agrupa diversos métodos y objetos que tienen relación con el sistema local. Contiene, entre otros, tres objetos ***static*** que son:

System.in: Objeto de la clase ***InputStream*** preparado para recibir datos desde la entrada estándar del sistema (habitualmente el teclado).

System.out: Objeto de la clase ***PrintStream*** que imprimirá los datos en la salida estándar del sistema (normalmente asociado con la pantalla).

System.err: Objeto de la clase ***PrintStream***. Utilizado para mensajes de error que salen también por pantalla por defecto.

Estas clases permiten la comunicación alfanumérica con el programa a través de lo métodos incluidos en la Tabla 9.4. Son métodos que permiten la entrada/salida a un nivel muy elemental.

Métodos de System.in	**Función que realizan**
int read()	Lee un carácter y lo devuelve como int.
Métodos de System.out y System.err	**Función que realizan**
int print(cualquier tipo)	Imprime en pantalla el argumento que se le pase. Puede recibir cualquier tipo primitivo de variable de Java.
int println(cualquier tipo)	Como el anterior, pero añadiendo '\n' (nueva línea) al final.

Tabla 9.4. Métodos elementales de lectura y escritura.

Existen tres métodos de ***System*** que permiten sustituir la entrada y salida estándar. Por ejemplo, se utiliza para hacer que el programa lea de un archivo y no del teclado.

```
System.setIn(InputStream is);
System.setOut(PrintStream ps);
System.setErr(PrintStream ps);
```

El argumento de ***setIn()*** no tiene que ser necesariamente del tipo ***InputStream***. Es una referencia a la clase base, y por tanto puede apuntar a objetos de cualquiera de sus clases derivadas (como ***FileInputStream***). Asimismo, el constructor de ***PrintStream*** acepta un ***OutputStream***, luego se puede dirigir la salida estándar a cualquiera de las clases definidas para salida.

Si se utilizan estas sentencias con un compilador de ***Java 1.1*** se obtiene un mensaje de método obsoleto (***deprecated***) al crear un objeto ***PrintStream***. Al señalar como obsoleto el constructor de esta clase se pretendía animar al uso de ***PrintWriter***, pero existen casos en los cuales es imprescindible un elemento ***PrintStream***. Afortunadamente, ***Java 1.2*** ha reconsiderado esta decisión y de nuevo se puede utilizar sin problemas.

9.2.1 Salida de texto y variables por pantalla

Para imprimir en la pantalla se utilizan los métodos ***System.out.print()*** y ***System.out.println()***. Son los primeros métodos que aprende cualquier programador. Sus características fundamentales son:

1. Pueden imprimir valores escritos directamente en el código o cualquier tipo de variable primitiva de ***Java***.

```
System.out.println("Hola, Mundo!");
System.out.println(57);
double numeroPI = 3.141592654;
System.out.println(numeroPI);
String hola = new String("Hola");
System.out.println(hola);
```

2. Se pueden imprimir varias variables en una llamada al método correspondiente utilizando el operador + de concatenación, que equivale a convertir a ***String*** todas las variables que no lo sean y concatenar las cadenas de caracteres (el primer argumento debe ser un ***String***).

```
System.out.println("Hola, Mundo! " + numeroPI);
```

Se debe recordar que los objetos ***System.out*** y ***System.err*** son de la clase ***PrintStream*** y aunque imprimen las variables de un modo legible, no permiten dar a la salida un formato a medida. El programador no puede especificar un formato distinto al disponible por defecto.

9.2.2 Lectura desde teclado

Para leer desde teclado se puede utilizar el método ***System.in.read()*** de la clase ***InputStream***. Este método lee un carácter por cada llamada. Su valor de retorno es un ***int***. Si se espera cualquier otro tipo hay que hacer una conversión explícita mediante un ***cast***.

```
char c;
c=(char)System.in.read();
```

Este método puede lanzar la excepción ***java.io.IOException*** y siempre habrá que ocuparse de ella, por ejemplo en la forma:

```
try {
    c=(char)System.in.read();
}
catch(java.io.IOException ioex) {
    // qué hacer cuando ocurra la excepción
}
```

Para leer datos más largos que un simple carácter es necesario emplear un bucle ***while*** o ***for*** y unir los caracteres. Por ejemplo, para leer una línea completa se podría utilizar un bucle ***while*** guardando los caracteres leídos en un ***String*** o en un ***StringBuffer*** (más rápido que ***String***):

```
char c;
String frase = new String("");          // StringBuffer frase=new StringBuffer("");
try {
    while((c=System.in.read()) != '\n')
    frase = frase + c;                  // frase.append(c);
}
catch(java.io.IOException ioex) {}
```

Una vez que se lee una línea, ésta puede contener números de coma flotante, etc. Sin embargo, hay una manera más fácil de conseguir lo mismo: utilizar adecuadamente la librería ***java.io***.

9.2.3 Método práctico para leer desde teclado

Para facilitar la lectura de teclado se puede conseguir que se lea una línea entera con una sola orden si se utiliza un objeto ***BufferedReader***. El método ***String readLine()*** perteneciente a ***BufferReader*** lee todos los caracteres hasta encontrar un '\n' o '\r' y los devuelve como un ***String*** (sin incluir '\n' ni '\r'). Este método también puede lanzar ***java.io.IOException***.

System.in es un objeto de la clase ***InputStream***. ***BufferedReader*** pide un ***Reader*** en el constructor. El puente de unión necesario lo dará ***InputStreamReader***, que acepta un ***InputStream*** como argumento del constructor y es una clase derivada de ***Reader***. Por lo tanto si se desea leer una línea completa desde la entrada estándar habrá que utilizar el siguiente código:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// o en una línea:
// BufferedReader br2 = new BufferedReader(new InputStreamReader(System.in));

String frase = br2.readLine();    // Se lee la línea con una llamada
```

Así ya se ha leido una línea del teclado. El thread que ejecute este código estará parado en esta línea hasta que el usuario termine la línea (pulse ***return***). Es más sencillo y práctico que la posibilidad anterior.

¿Y qué hacer con una línea entera? La clase ***java.util.StringTokenizer*** da la posibilidad de separar una cadena de carácteres en las “palabras” (***tokens***) que la forman (por defecto, conjuntos de

caracteres separados por un espacio, '\t', '\r', o por '\n'). Cuando sea preciso se pueden convertir las "palabras" en números.

La Tabla 9.5 muestra los métodos más prácticos de la clase ***StringTokenizer***.

Métodos	**Función que realizan**
StringTokenizer(String)	Constructor a partir de la cadena que hay que separar
boolean hasMoreTokens()	¿Hay más palabras disponibles en la cadena?
String nextToken()	Devuelve el siguiente token de la cadena
int countTokens()	Devuelve el número de tokens que se pueden extraer de la frase

Tabla 9.5. Métodos de StringTokenizer.

La clase ***StreamTokenizer*** de ***java.io*** aporta posibilidades más avanzadas que ***StringTokenizer***, pero también es más compleja. Directamente separa en ***tokens*** lo que entra por un ***InputStream*** o ***Reader***.

Se recuerda que la manera de convertir un ***String*** del tipo "3.141592654" en el valor ***double*** correspondiente es crear un objeto ***Double*** a partir de él y luego extraer su valor ***double***:

```
double pi = (Double.valueOf("3.141592654")).doubleValue();
```

El uso de estas clases facilita el acceso desde teclado, resultando un código más fácil de escribir y de leer. Además tiene la ventaja de que se puede generalizar a la lectura de archivos.

9.3 LECTURA Y ESCRITURA DE ARCHIVOS

Aunque el manejo de archivos tiene características especiales, se puede utilizar lo dicho hasta ahora para las entradas y salidas estándar con pequeñas variaciones. ***Java*** ofrece las siguientes posibilidades:

Existen las clases ***FileInputStream*** y ***FileOutputStream*** (extendiendo ***InputStream*** y ***OutputStream***) que permiten leer y escribir ***bytes*** en archivos. Para archivos de texto son preferibles ***FileReader*** (desciende de ***Reader***) y ***FileWriter*** (desciende de ***Writer***), que realizan las mismas funciones. Se puede construir un objeto de cualquiera de estas cuatro clases a partir de un ***String*** que contenga el nombre o la dirección en disco del archivo o con un objeto de la clase ***File*** que representa dicho archivo. Por ejemplo el código

```
FileReader fr1 = new FileReader("archivo.txt");
```

es equivalente a:

```
File f = new File("archivo.txt");
FileReader fr2 = new FileReader(f);
```

Si no encuentran el archivo indicado, los constructores de ***FileReader*** y ***FileInputStream*** pueden lanzar la excepción ***java.io.FileNotFoundException***.

Los constructores de ***FileWriter*** y ***FileOutputStream*** pueden lanzar ***java.io.IOException***. Si no encuentran el archivo indicado, lo crean nuevo. Por defecto, estas dos clases comienzan a escribir al comienzo del archivo. Para escribir detrás de lo que ya existe en el archivo ("append"), se utiliza un segundo argumento de tipo ***boolean*** con valor ***true***:

```
FileWriter fw = new FileWriter("archivo.txt", true);
```

Las clases que se explican a continuación permiten un manejo más fácil y eficiente que las vistas hasta ahora.

9.3.1 Clases File y FileDialog

Un objeto de la clase ***File*** puede representar un ***archivo*** o un ***directorio***. Tiene los siguientes constructores:

```
File(String name)
File(String dir, String name)
File(File dir, String name).
```

Se puede dar el nombre de un archivo, el nombre y el directorio, o sólo el directorio, como ***path*** absoluto y como ***path*** relativo al directorio actual. Para saber si el archivo existe se puede llamar al método ***boolean exists().***

```
File f1 = new File("c:\\windows\\notepad.exe");  // La barra '\' se escribe '\\'
File f2 = new File("c:\\windows");               // Un directorio
File f3 = new File(f2, "notepad.exe");           // Es igual a f1
```

Si ***File*** representa un archivo que existe los métodos de la Tabla 9.6 dan información de él.

Métodos	**Función que realizan**
boolean isFile()	true si el archivo existe
long length()	tamaño del archivo en bytes
long lastModified()	fecha de la última modificación
boolean canRead()	true si se puede leer
boolean canWrite()	true si se puede escribir
delete()	borrar el archivo
RenameTo(File)	cambiar el nombre

Tabla 9.6. Métodos de File para archivos.

Si representa un directorio se pueden utilizar los de la Tabla 9.7:

Métodos	**Función que realizan**
boolean isDirectory()	true si existe el directorio
mkdir()	crear el directorio
delete()	borrar el directorio
String[] list()	devuelve los archivos que se encuentran en el directorio

Tabla 9.7. Métodos de File para directorios.

Por último, otros métodos incluidos en la Tabla 9.8 devuelven el ***path*** del archivo de distintas maneras.

Métodos	Función que realizan
String getPath()	Devuelve el path que contiene el objeto File
String getName()	Devuelve el nombre del archivo
String getAbsolutePath()	Devuelve el path absoluto (juntando el relativo al actual)
String getParent()	Devuelve el directorio padre

Tabla 9.8. Métodos de File que devuelven el path.

Una forma típica de preguntar por un archivo es presentar un caja de diálogo. La clase ***java.awt.FileDialog*** presenta el diálogo típico de cada sistema operativo para guardar o abrir ficheros. Sus constructores son:

```
FileDialog(Frame fr)
FileDialog(Frame fr, String title)
FileDialog(Frame fr, String title, int type)
```

donde ***type*** puede ser ***FileDialog.LOAD*** o ***FileDialog.SAVE*** según la operación que se desee realizar.

Es muy fácil conectar este diálogo con un ***File***, utilizando los métodos ***String getFile()*** y ***String getDirectory()***. Por ejemplo:

```
FileDialog fd = new FileDialog(f, "Elija un archivo");
fd.show();
File f = new File(fd.getDirectory(), fd.getFile());
```

9.3.2 Lectura de archivos de texto

Se puede crear un objeto ***BufferedReader*** para leer de un archivo de texto de la siguiente manera:

```
BufferedReader br = new BufferedReader(new FileReader("archivo.txt"));
```

Utilizando el objeto de tipo ***BufferedReader*** se puede conseguir exactamente lo mismo que en las secciones anteriores utilizando el método ***readLine()*** y la clase ***StringTokenizer***. En el caso de archivos es muy importante utilizar el ***buffer*** puesto que la tarea de escribir en disco es muy lenta respecto a los procesos del programa y realizar las operaciones de lectura de golpe y no de una en una hace mucho más eficiente el acceso. Por ejemplo:

```
// Lee un archivo entero de la misma manera que de teclado
String texto = new String();
try {
    FileReader fr = new FileReader("archivo.txt");
    entrada = new BufferedReader(fr);
    String s;
    while((s = entrada.readLine()) != null)
        texto += s;
    entrada.close();
}
catch(java.io.FileNotFoundException fnfex) {
    System.out.println("Archivo no encontrado: " + fnfex);}
catch(java.io.IOException ioex) {}
```

9.3.3 Escritura de archivos de texto

La clase ***PrintWriter*** es la más práctica para escribir un archivo de texto porque posee los métodos ***print***(cualquier tipo) y ***println***(cualquier tipo), idénticos a los de ***System.out*** (de clase ***PrintStream***).

Un objeto ***PrintWriter*** se puede crear a partir de un ***BufferedWriter*** (para disponer de ***buffer***), que se crea a partir del ***FileWriter*** al que se la pasa el nombre del archivo. Después, escribir en el archivo es tan fácil como en pantalla. El siguiente ejemplo ilustra lo anterior:

```
try {
   FileWriter fw = new FileWriter("escribeme.txt");
   BufferedWriter bw = new BufferedWriter(fw);
   PrintWriter salida = new PrintWriter(bw);
   salida.println("Hola, soy la primera línea");
   salida.close();
   // Modo append
   bw = new BufferedWriter(new FileWriter("escribeme.txt", true));
   salida = new PrintWriter(bw);
   salida.print("Y yo soy la segunda. ");
   double b = 123.45;
   salida.println(b);
   salida.close();
}
cacth(java.io.IOException ioex) { }
```

9.3.4 Archivos que no son de texto

DataInputStream y ***DataOutputStream*** son clases de ***Java 1.0*** que no han sido alteradas hasta ahora. Para leer y escribir datos primitivos directamente (sin convertir a/de ***String***) siguen siendo más útiles estas dos clases.

Son clases diseñadas para trabajar de manera conjunta. Una puede leer lo que la otra escribe, que en sí no es algo legible, sino el dato como una secuencia de ***bytes***. Por ello se utilizan para almacenar datos de manera independiente de la plataforma (o para mandarlos por una red entre ordenadores muy distintos).

El problema es que obligan a utilizar clases que descienden de ***InputStream*** y ***OutputStream*** y por lo tanto algo más complicadas de utilizar. El siguiente código primero escribe en el fichero *prueba.dat* para después leer los datos escritos:

```
// Escritura de una variable double
DataOutputStream dos = new DataOutputStream(
   new BufferedOutputStream(
      new FileOutputStream("prueba.dat")));
double d1 = 17/7;
dos.writeDouble(d);
dos.close();
// Lectura de la variable double
DataInputStream dis = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("prueba.dat")));
double d2 = dis.readDouble();
```

9.4 SERIALIZACIÓN

La ***serialización*** es un proceso por el que un objeto cualquiera se puede convertir en una ***secuencia de bytes*** con la que más tarde se podrá reconstruir dicho objeto manteniendo el valor de sus variables. Esto permite guardar un objeto en un archivo o mandarlo por la red.

Para que una clase pueda utilizar la serialización, debe implementar la interface ***Serializable***, que no define ningún método. Casi todas las clases estándar de ***Java*** son serializables. La clase MiClase se podría serializar declarándola como:

```
public class MiClase implements Serializable { }
```

Para escribir y leer objetos se utilizan las clases ***ObjectInputStream*** y ***ObjectOutputStream***, que cuentan con los métodos ***writeObject()*** y ***readObject()***. Por ejemplo:

```
ObjectOutputStream objout = new ObjectOutputStream(
                                new FileOutputStream("archivo.x"));
String s = new String("Me van a serializar");
objout.writeObject(s);
ObjectInputStream objin = new ObjectInputStream(new FileInputStream("archivo.x"));
String s2 = (String)objin.readObject();
```

Es importante tener en cuenta que ***readObject()*** devuelve un ***Object*** sobre el que se deberá hacer un ***casting*** para que el objeto sea útil. La reconstrucción necesita que el archivo ****.class*** esté al alcance del programa (como mínimo para hacer este ***casting***).

Al serializar un objeto, automáticamente se serializan todas sus variables y objetos miembro. A su vez se serializan los que estos objetos miembro puedan tener (todos deben ser serializables). También se reconstruyen de igual manera. Si se serializa un ***Vector*** que contiene varios ***Strings***, todo ello se convierte en una serie de ***bytes***. Al recuperarlo la reconstrucción deja todo en el lugar en que se guardó.

Si dos objetos contienen una referencia a otro, éste no se duplica si se escriben o leen ambos del mismo ***stream***. Es decir, si el mismo ***String*** estuviera contenido dos veces en el ***Vector***, sólo se guardaría una vez y al recuperarlo sólo se crearía un objeto con dos referencias contenidas en el vector.

9.4.1 Control de la serialización

Aunque lo mejor de la serialización es que su comportamiento automático es bueno y sencillo, existe la posibilidad de especificar cómo se deben hacer las cosas.

La palabra clave ***transient*** permite indicar que un objeto o varible miembro no sea serializado con el resto del objeto. Al recuperarlo, lo que esté marcado como ***transient*** será ***0***, ***null*** o ***false*** (en esta operación no se llama a ningún constructor) hasta que se le dé un nuevo valor. Podría ser el caso de un ***password*** que no se quiere guardar por seguridad.

Las variables y objetos ***static*** no son serializados. Si se quieren incluir hay que escribir el código que lo haga. Por ejemplo, habrá que programar un método que serialice los objetos estáticos al que se llamará después de serializar el resto de los elementos. También habría que recuperarlos explícitamente después de recuperar el resto de los objetos.

Las clases que implementan ***Serializable*** pueden definir dos métodos con los que controlar la serialización. No están obligadas a hacerlo porque una clase sin estos métodos obtiene directamente el comportamiento por defecto. Si los define serán los que se utilicen al serializar:

```
private void writeObject(ObjectOutputStream stream) throws IOException
private void readObject(ObjectInputStream stream) throws IOException
```

El primero permite indicar qué se escribe o añadir otras instrucciones al comportamiento por defecto. El segundo debe poder leer lo que escribe ***writeObject()***. Puede usarse por ejemplo para poner al día las variables que lo necesiten al ser recuperado un objeto. Hay que leer en el mismo orden en que se escribieron los objetos.

Se puede obtener el comportamiento por defecto dentro de estos métodos llamando a ***stream.defaultWriteObject()*** y ***stream.defaultReadObject()***.

Para guardar explícitamente los tipos primitivos se puede utilizar los métodos que proporcionan ***ObjectInputStream*** y ***ObjectOutputStream***, idénticos a los de ***DataInputStream*** y ***DataOutputStream*** (***writeInt()***, ***readDouble()***, ...) o guardar objetos de sus clases equivalentes (***Integer***, ***Double***...).

Por ejemplo, si en una clase llamada ***Tierra*** se necesita que al serializar un objeto siempre le acompañe la constante ***g*** (9,8) definida como ***static*** el código podría ser:

```
static double g = 9.8;

private void writeObject(ObjectOutputStream stream) throws IOException {
      stream.defaultWriteObject();
      stream.writeDouble(g);
}

private void readObject(ObjectInputStream stream) throws IOException {
      stream.defaultReadObject();
      g = stream.readDouble(g);
}
```

9.4.2 Externalizable

La interface ***Externalizable*** extiende ***Serializable***. Tiene el mismo objetivo que ésta, pero no tiene ningún comportamiento automático, todo se deja en manos del programador.

Externalizable tiene dos métodos que deben implementarse.

```
interface Externalizable {
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
                                                    ClassNotFoundException;
}
```

Al transformar un objeto, el método ***writeExternal()*** es responsable de todo lo que se hace. Sólo se guardará lo que dentro de éste método se indique.

El método ***readExternal()*** debe ser capaz de recuperar lo guardado por ***writeExternal()***. La lectura debe ser en el mismo orden que la escritura. Es importante saber que antes de llamar a este método se llama al constructor por defecto de la clase.

Como se ve el comportamiento de ***Externalizable*** es muy similar al de ***Serializable***.

9.5 LECTURA DE UN ARCHIVO EN UN SERVIDOR DE INTERNET

Teniendo la dirección de Internet de un archivo, la librería de ***Java*** permite leer este archivo utilizando un ***stream***. Es una aplicación muy sencilla que muestra la polivalencia del concepto de ***stream***.

En el package ***java.net*** existe la clase ***URL***, que representa una dirección de Internet. Esta clase tiene el método ***InputStream openStream(URL dir)*** que abre un ***stream*** con origen en la dirección de Internet.

A partir de ahí, se trata como cualquier elemento ***InputStream***. Por ejemplo:

```
//Lectura del archivo (texto HTML)
URL direccion = new URL("http://www1.ceit.es/subdir/MiPagina.htm");
```

```
String s = new String();
String html = new String();
try {
     BufferedReader br = new BufferedReader(
                              new InputStreamReader(
                                  direccion.openStream()));
     while((s = br.readLine()) != null)
          html += s + '\n';
     br.close();
}
catch(Exception e) {
     System.err.println(e);
}
```

10. OTRAS CAPACIDADES DE JAVA

A lo largo de este manual se han presentado algunos de los fundamentos del lenguaje ***Java***. Debido a que ***Java*** engloba en el propio lenguaje muchos de los conceptos de la informática moderna la inclusión de todos ellos sobrepasaría ampliamente el carácter introductorio de este manual.

No se puede sin embargo finalizar esta introducción al lenguaje ***Java*** sin una breve descripción de algunas de las capacidades más interesantes del lenguaje.

10.1 JAVA FOUNDATION CLASSES (JFC) Y JAVA 2D

Las ***JFC***, ***Java™ Foundation Classes*** son un conjunto de componentes y características para ayudar a construir los entornos gráficos de los programas o GUIs (*Graphical User Interfaces*). Incluye prácticamente todo tipo de elementos gráficos como botones, paneles, menús y ventanas, con muchas ventajas sobre el AWT.

Swing es una parte de las ***JFC*** que permite incorporar en las aplicaciones elementos gráficos de una forma mucho más versátil y con más capacidades que utilizando el **AWT** básico de ***Java***. Algunas de las características más interesantes son:

1. Cualquier programa que utiliza componentes de ***Swing*** puede elegir el aspecto que desea para sus ventanas y elementos gráficos: entorno ***Windows 95/98/NT***, entorno ***Motif*** (asociado a sistemas UNIX) o ***Metal*** (aspecto propio de ***Java***, común a todas las plataformas).
2. Cualquier componente gráfico de ***Swing*** presenta más propiedades que el correspondiente elemento del ***AWT***: Los botones pueden incorporan imágenes, hay nuevos ***layouts*** y paneles, menús, …
3. Posibilidad de ***Drag & Drop***, es decir de seleccionar componentes con el ratón y arrastrar a otro lugar de la pantalla.

En la versión ***JDK 1.2*** se incorpora como parte de las JFC el llamado ***Java 2D***, que permite a los desarrolladores incorporar texto, imágenes y gráficos en dos dimensiones de gran calidad. Además da soporte para poder imprimir documentos complejos.

A partir de la versión ***1.2*** de ***Java*** las ***JFC*** forman parte del propio ***JDK***. Si se desea utilizar desde la versión ***1.1*** es necesario instalar las ***JFC*** de forma independiente.

10.2 JAVA MEDIA FRAMEWORK (JMF)

El API ***JMF*** (***Java Media FrameWork***) especifica una arquitectura, un protocolo de transmisión de datos y unos elementos gráficos simples y unificados para la reproducción de contenidos ***multimedia***, esto es vídeo, audio y animaciones, principalmente.

Los distintos JDK aparecidos hasta la publicación de este manual no incorporan este API de JMF. Es necesario instalar un software que complementa el JDK.

10.3 JAVA 3D

El API de ***Java 3D™*** es un conjunto de clases para crear aplicaciones y ***applets*** con elementos ***3D***. Ofrece a los desarrolladores la posibilidad de manipular geometrías complejas en tres dimensiones. La principal ventaja que presenta este API 3D frente a otros entornos de programación 3D es que permite crear aplicaciones gráficas 3D independientes del tipo de sistema.

Java 3D es un conjunto de clases, interfaces y librerías de alto nivel que permiten aprovechar la aceleración gráfica por hardware que incorporan muchas tarjetas gráficas, ya que las llamadas a los métodos de ***Java 3D*** son transformadas en llamadas a funciones de ***OpenGL*** o ***Direct3D***.

Aunque tanto conceptualmente como oficialmente ***Java 3D*** forma parte del API ***JMF***, se trata de unas librerías que se instalan independientemente del ***JMF***.

10.4 JAVABEANS

El API de ***JavaBeans*** hace posible escribir "componentes de software" en el lenguaje ***Java***. Los ***componentes*** son elementos reutilizables que pueden incorporarse gráficamente a otros componentes como ***applets*** y aplicaciones utilizando herramientas gráficas de desarrollo.

Cada componente ofrece sus características concretas (por ejemplo sus métodos públicos y sus eventos) a los entornos gráficos de desarrollo permitiendo su manipulación visual. Son análogos a otros componentes de algunos entornos visuales, como por ejemplo los controles de Visual Basic.

El ***BDK*** (***Beans Developer Kit***) es un conjunto de herramientas para desarrollar ***JavaBeans***. Se trata de un kit no incorporado en los distintos ***JDK*** de ***Java***.

10.5 JAVA EN LA RED

A diferencia de otros lenguajes de programación, ***Java*** presenta de una forma estándar para todas las plataformas y sistemas operativos, un conjunto de clases que permiten la comunicación entre aplicaciones que se ejecutan en distintos ordenadores.

El package ***java.net*** del API de ***Java*** incluye las clases necesarias para establecer conexiones, crear servidores, enviar y recibir datos, y para el resto de operaciones utilizadas en las comunicaciones a través de redes de ordenadores. Existen además otros APIs independientes preparados especialmente para realizar unas determinadas tareas, como son ***Servlets, RMI*** y ***Java IDL***. Muchos de estos APIs utilizan internamente las clases presentes en ***java.net***.

10.6 JAVA EN EL SERVIDOR: SERVLETS

Los ***Servlets*** son módulos que permiten sustituir o utilizar el lenguaje ***Java*** en lugar de los programas ***CGI*** escritos en otros lenguajes como C/C++ o ***Perl***. Los programas ***CGI*** son aplicaciones que se ejecutan en un ***servidor Web*** en respuesta a una acción de un browser remoto (petición de una página HTML, envío de los datos de un formulario, etc.). Permiten generar páginas HTML dinámicas, esto es, páginas HTML cuyo contenido puede variar y que por lo tanto no pueden almacenarse en un fichero en el servidor.

Los ***Servlets*** no tienen entorno gráfico ya que se ejecutan en el servidor. Reciben unos datos y su salida o respuesta son principalmente ficheros de texto HTML.

Los ***servlets*** son desarrollados utilizando el API ***Java Servlet***, que es una extensión de ***Java*** que hasta la fecha no forma parte de ninguno de los JDK. Es necesario instalar el software específico de ***Java Servlet***.

10.7 RMI Y JAVA IDL

Tanto ***RMI*** (***Remote Method Invocation***) como ***Java IDL*** (***Java Interface Definition Language***) son herramientas para desarrollar ***aplicaciones distribuidas***. Estas aplicaciones presentan la característica de que una aplicación puede ejecutar funciones y métodos en varios ordenadores distintos. Utilizando una referencia a un objeto que se encuentra en un ordenador remoto, es posible ejecutar métodos de ese objeto desde una aplicación que se ejecuta en un ordenador distinto. ***RMI y Java IDL*** proporcionan los mecanismos mediante los cuales los distintos objetos distribuidos se comunican y se transmiten la información. Son por lo tanto tecnologías que permiten la creación y uso de ***objetos distribuidos***, esto es, objetos o programas que interactúan en diferentes plataformas y ordenadores a través de una red.

RMI es una solución basada íntegramente en ***Java***. Incorpora métodos para localizar los objetos remotos, comunicarse con ellos e incluso enviar un objeto de ***Java***, "por valor", de un objeto distribuido a otro.

Java IDL permite la conectividad entre objetos distribuidos utilizando ***CORBA (Common Object Request Broker Architecture***). ***CORBA*** es un estándar para la interconexión entre objetos distribuidos. Existen implementaciones de ***CORBA*** en varios lenguajes, lo que posibilita comunicar objetos realizados en distintos lenguajes como ***Java***, ***C/C++***, ***COBOL***, …

Tanto ***RMI*** como ***Java IDL*** están incluidos en el ***JDK 1.2*** de ***Sun***. En el caso de ***Java IDL*** se precisa de una utilidad adicional (llamada ***idltojava***) que genera el código necesario para comunicarse con cualquier implementación ***CORBA***.

10.8 SEGURIDAD EN JAVA

El espacio natural de trabajo de la Informática moderna y por lo tanto del lenguaje ***Java*** son las redes de ordenadores y en especial ***Internet***. En una red como ***Internet***, utilizada por millones de usuarios, la seguridad adquiere una importancia vital.

Desde su aparición ***Java*** ha ido incorporando elementos destinados a proporcionar un mayor control sobre la seguridad de los datos y programas enviados a través de una red. Los distintos ***JDK*** incorporan herramientas para añadir seguridad a las aplicaciones ***Java***: *firmas digitales*, *transmisión segura de datos, ...* ***Java*** permite establecer distintos niveles de seguridad, lo que posibilita una gran flexibilidad para asignar o denegar permisos. Existe abundante información sobre estas herramientas que el lector deberá consultar en caso necesario.

10.9 ACCESO A BASES DE DATOS (JDBC)

JDBC (***Java DataBase Connectivity***) es el estándar de ***Java*** para conectarse con bases de datos. Se estima que aproximadamente la mitad del software que se crea en la actualidad incorpora

operaciones de lectura y escritura con bases de datos. ***JDBC*** está diseñado para ser independiente de la plataforma e incluso de la base de datos sobra la que se desee actuar.

Para conseguir esta independencia, ***JDBC*** ofrece un sistema estándar de interconexión con las bases de datos, muy similar al ***SQL*** (***Structured Query Language***). Los distintos vendedores de bases de datos crean los elementos necesarios que actúan como puente entre ***JDBC*** y la propia base de datos.

La versión ***JDBC 1.0*** forma parte del ***JDK 1.1***. Después de distintas revisiones, actualmente ha aparecido la versión ***JDBC 2.0***, incluida en el ***JDK 1.2***.

10.10 JAVA NATIVE INTERFACE (JNI)

JNI (***Java Native Interface***) es el interface de programación de ***Java*** para ejecutar código nativo, es decir código compilado al lenguaje binario propio de una plataforma o sistema de ordenador. Se incluye en el ***JDK*** las herramientas necesarias para su utilización.

JNI permite al código de ***Java*** que se ejecuta dentro de la ***JVM*** interactuar con aplicaciones y librerías escritas en otros lenguajes, como ***C/C++*** o incluso lenguaje ***ensamblador***. Incorpora a su vez las herramientas para ejecutar código ***Java*** desde aplicaciones desarrolladas en otros lenguajes. El entorno ***JNI*** ofrece por lo tanto a los métodos nativos utilizar objetos de ***Java*** de igual forma que el código ***Java*** puede utilizar estos objetos nativos. Tanto la parte de ***Java*** como la parte nativa de una aplicación pueden crear, actualizar y acceder a los objetos programados en ***Java*** y compartir dichos objetos.