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 de 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.ArrayList;
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 ***ArrayList*** simplemente con el nombre ***ArrayList*** en lugar de con el nombre completo ***java.util.ArrayList***, 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 ***ArrayList***). 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 17, 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 (`ArrayList v = new ArrayList();`) 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 objeto ***v*** de la clase ***ArrayList***, utilizando el método ***add()*** de la propia clase ***ArrayList***.

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.

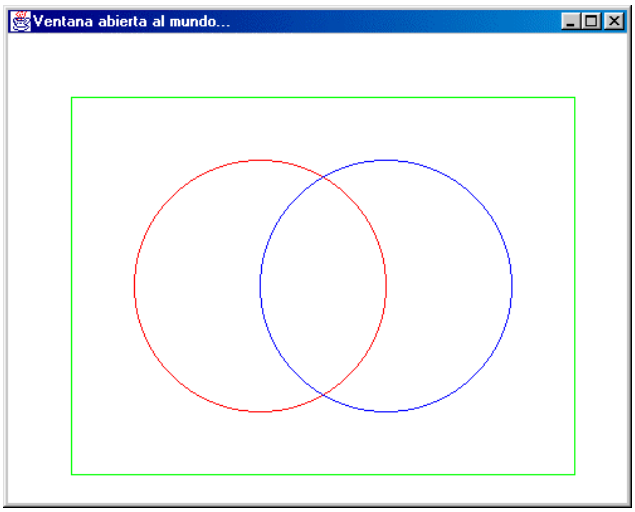

Figura 1.1. Resultado de la ejecución del Ejemplo1.

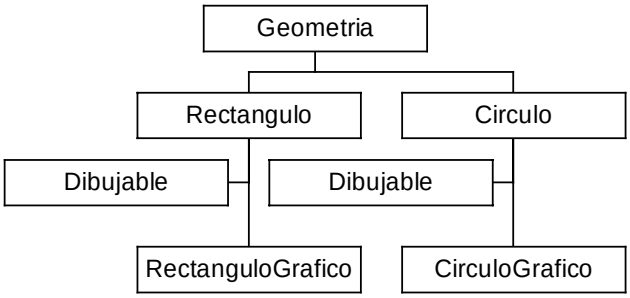

Figura 1.2. Jerarquía de clases utilizadas.

1.3.2 Clase Geometria

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 **π** 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
```