

24

Introducción a las aplicaciones y a los applets de Java

Objetivos

- Escribir aplicaciones sencillas con Java.
- Utilizar instrucciones de entrada y salida.
- Observar algunas de las excitantes capacidades de Java a través de varios applets de demostración proporcionados con el Java 2 Software Development Kit.
- Comprender la diferencia entre un applet y una aplicación.
- Escribir applets sencillos en Java.
- Escribir archivo sencillos en Lenguaje de Marcación de Hipertexto (HTML) para cargar un applet en el **applet viewer** o en un navegador de la World Wide Web.

Los comentarios son libres, pero los hechos son sagrados.

C. P. Scott

El acreedor tiene mejor memoria que el deudor.

James Howell

*Cuando tengo que tomar una decisión, siempre me pregunto,
“¿qué sería lo más divertido?”*

Peggy Walker

Unas clases fracasan, otras triunfan, y otras son eliminadas.

Mao Tse Tung



Plan general

- 24.1 Introducción**
- 24.2 Fundamentos de un entorno típico de Java**
- 24.3 Notas generales acerca de Java y de este libro**
- 24.4 Un programa sencillo: Impresión de una línea de texto**
- 24.5 Otra aplicación en Java: Suma de enteros**
- 24.6 Applets de ejemplo del Java 2 Software Development Kit**
 - 24.6.1 El applet Tictactoe**
 - 24.6.2 El applet Drawtest**
 - 24.6.3 El applet Java2D**
- 24.7 Un applet sencillo en Java: Cómo dibujar una cadena**
- 24.8 Dos ejemplos más de applets: Cómo dibujar cadenas y líneas**
- 24.9 Otro applet de Java: Suma de enteros**

Resumen • Terminología • Errores comunes de programación • Tips para prevenir errores • Buenas prácticas de programación • Tip de rendimiento • Tips de portabilidad • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

24.1 Introducción

Ahora procederemos a estudiar Java, un poderoso lenguaje orientado a objetos, divertido para los novatos, pero también apropiado para los programadores experimentados en la construcción de sistemas de información importantes. Java seguramente será la elección del nuevo milenio para la implementación de aplicaciones basadas en Internet e Intranets, así como el software para dispositivos que se comunican entre redes (tales como teléfonos celulares, paginadores y asistentes digitales personales). ¡No se sorprenda cuando su nuevo estéreo y otros dispositivos en su casa se conecten en red por medio de tecnología Java!

En los capítulos correspondientes a C de este libro presentamos un tratamiento de la programación por procedimientos y el diseño de programas arriba-abajo. En los capítulos de C++, presentamos paradigmas adicionales de programación; programación basada en objetos (con clases, encapsulamiento, objetos, y sobrecarga de operadores), programación orientada a objetos (con herencia y polimorfismo) y programación genérica (con plantillas de funciones y plantillas de clases). Estos paradigmas de programación son cruciales para el desarrollo de sistemas de software elegante, robusto y de fácil mantenimiento. En los capítulos de Java explicamos los gráficos, las interfaces gráficas de usuario, multimedia y la programación orientada a eventos: Sun Microsystems desarrolló Java, teniendo en mente estas populares tecnologías.

Dominar estos variados paradigmas de desarrollo y las tecnologías que explicamos en el libro le ayudará a construir fundamentos sólidos de programación. Trabajamos duro para crear lo que esperamos será una experiencia informativa, entretenida y desafiante para usted.

Una implementación de Java está disponible en el sitio Web de Java

java.sun.com

Estos capítulos están basados en la versión de Java más reciente de Sun, la *Java 2 Platform*. Sun proporciona una implementación de *Java 2 Platform*, llamada *Java 2 Software Development Kit (J2SDK)*, versión 1.4 que incluye las herramientas que usted necesita para escribir software en Java. La extraordinaria portabilidad de Java significa que los programas de este libro funcionarán correctamente en cualquier versión de J2SDK 1.4.

En los capítulos 24 a 30, presentamos la programación en Java a una profundidad razonable para un libro introductorio como éste. Usted aprenderá a crear programas en Java llamados aplicaciones y applets; las principales diferencias entre Java, C y C++; la programación orientada y basada en objetos en Java; la programación de gráficos con una variedad de colores, fuentes, contornos de figuras, y formas rellenas; la programación de interfaces gráficas de usuario (GUIs) con los componentes Swing de Java; y la programación multimedia con efectos tales como clips de audio, procesamiento de imágenes, mapas de imágenes y animación.

24.2 Fundamentos de un entorno típico de Java

Por lo general, los sistemas en Java constan de diversas partes: un ambiente, el lenguaje, la interfaz de programación de aplicaciones de Java (API) y varias bibliotecas de clases. La siguiente explicación expone un entorno de programación típico de Java como lo muestra la figura 24.1.

Los programas en Java generalmente pasan a través de cinco fases para poder ejecutarse (figura 24.1). Éstas son: *edición*, *compilación*, *carga*, *verificación* y *ejecución*. Si usted no utiliza UNIX, Windows 95/98 o Windows NT, consulte los manuales para el ambiente Java de su sistema, o pregunte a su profesor cómo llevar a cabo estas tareas en su entorno particular (lo que probablemente será similar al entorno de la figura 24.1).

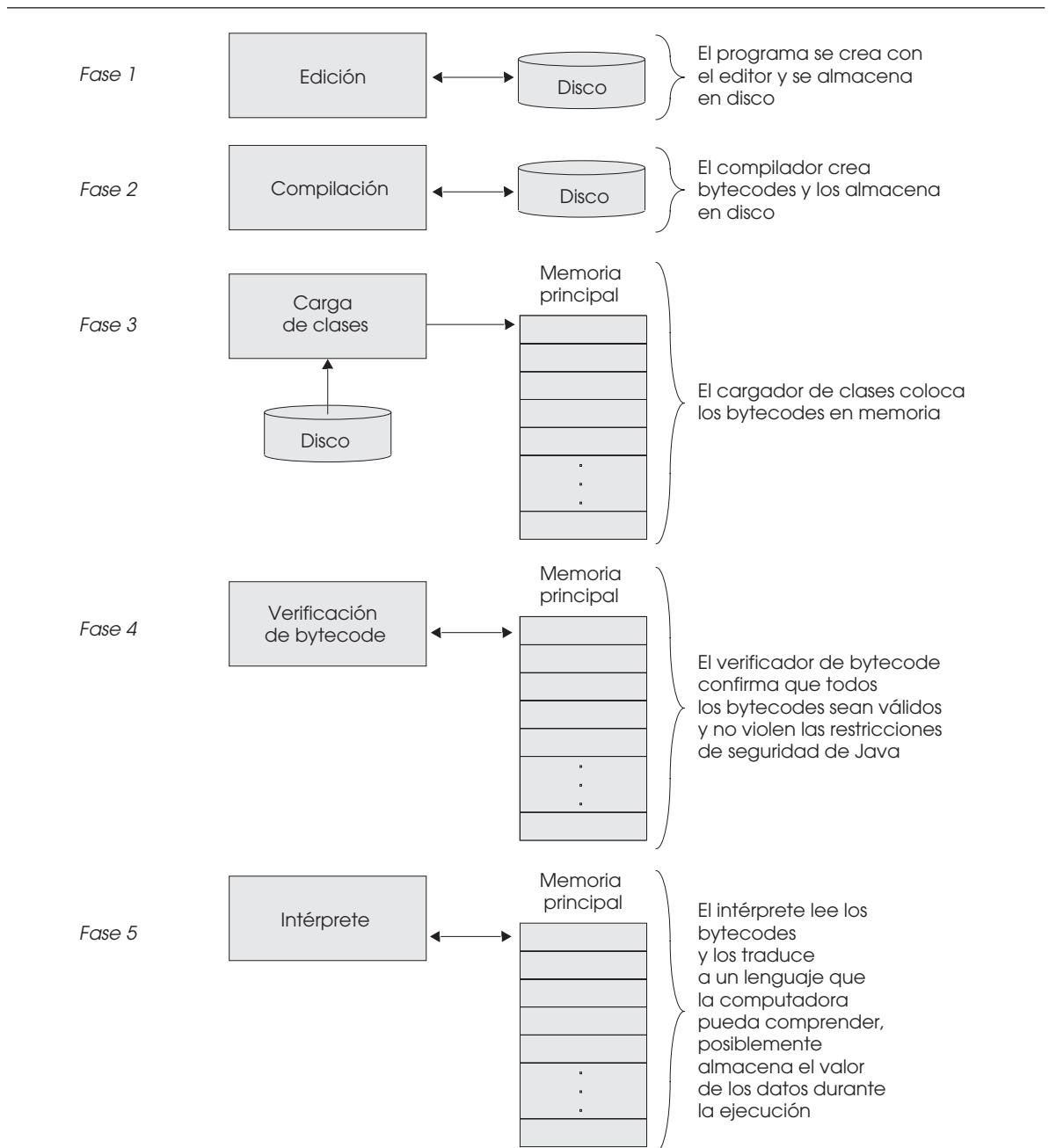


Figura 24.1 Un típico ambiente de Java.

La fase 1 consiste en editar el archivo. Esto se lleva a cabo con un *programa de edición*. El programador escribe un programa en Java por medio del editor y lo corrige si es necesario. Cuando el programador especifica que debe guardarse el archivo que se encuentra en el editor, el programa se almacena en un dispositivo de memoria secundaria tal como un disco. El archivo con el programa en Java termina con la extensión **.java**. Dos editores ampliamente utilizados en sistemas UNIX son **vi** y emacs. En Windows 95/98 y Windows NT programas de edición sencillos como el comando Edit de MS-DOS y el bloc de notas de Windows serán suficientes. Los ambientes integrados de desarrollo (IDEs) tales como Forte para Java de Sun, JBuilder de Borland, Visual Café de Symantec y Visual J++ de Microsoft tienen editores incluidos que se integran suavemente en el ambiente de programación. Asumimos que el lector sabe cómo editar un archivo.

En la fase 2, el programador aplica el comando **javac** para *compilar* el programa. El compilador de Java traduce el programa en Java a *bytecodes*, el lenguaje que comprende el intérprete de Java. Para compilar un programa llamado **Bienvenido.java**, escriba

```
javac Bienvenido.java
```

en la ventana de comando de su sistema (es decir, en el indicador de MS-DOS en Windows 95/98 y Windows NT, o en el indicador del shell en UNIX). Si el programa se compila correctamente, se produce un archivo **Bienvenido.class**. Éste es el archivo que contiene los bytecodes que se interpretarán durante la fase de ejecución.

La fase 3 se llama de *carga*. Esto se hace por medio del *cargador de clases*, el cual toma el archivo (o archivos) **.class** que contiene los bytecodes y los transfiere a la memoria. El archivo **.class** puede cargarse desde un disco en su sistema o sobre una red (tal como la red de su universidad o de su trabajo, o incluso por Internet). Existen dos tipos de programas para los cuales el cargador de clases carga archivos **.class**: *aplicaciones* y *applets*. Una aplicación Java es un programa tal como un procesador de palabras, una hoja de cálculo, un programa de dibujo, un programa de correo electrónico, etcétera, que por lo general se almacena y se ejecuta en memoria desde la computadora local del usuario. Un applet de Java es un pequeño programa que por lo general se almacena en una computadora remota que los usuarios conectan mediante un navegador de la World Wide Web. Los applets se cargan desde una computadora remota en el navegador, se ejecuta en el navegador y se descarta al completar la ejecución. Para ejecutar nuevamente un applet, el usuario debe apuntar su navegador a la ubicación apropiada en la World Wide Web y recargar el programa dentro del navegador.

Las aplicaciones se cargan en memoria y se ejecutan por medio del *intérprete de Java* con el comando **java**. Cuando se ejecuta una aplicación de Java llamada **Bienvenido**, el comando

```
java Bienvenido
```

invoca al intérprete para la aplicación **Bienvenido** y provoca que el cargador de clases cargue la información utilizada en el programa **Bienvenido**.

El cargador de clases también se ejecuta cuando se carga un applet de Java dentro de un navegador de la World Wide Web como *Netscape Communicator*, *Internet Explorer* de Microsoft, o *HotJava* de Sun. Los navegadores se utilizan para visualizar documentos de la World Wide Web llamados documentos *HTML* (*Lenguaje de Marcación de Hipertexto*). El HTML se utiliza para dar formato a un documento, de modo que sea fácil de comprender por la aplicación de navegador (nos introduciremos en HTML en la sección 24.7; para un tratamiento detallado de HTML y otras tecnologías de programación en Internet, revise nuestro libro *Internet and the World Wide Web How to program*). Un documento HTML puede hacer referencia a un applet de Java. Cuando el navegador ve un applet al que se hace referencia dentro de un documento HTML, el navegador lanza el cargador de clases de Java para cargar el applet (por lo general, desde la ubicación en donde se almacena el documento HTML). Los navegadores que soportan Java contienen un intérprete de Java. Una vez que se carga el applet, el intérprete de Java del navegador ejecuta el applet. Además, los applets pueden ejecutarse desde la línea de comando usando el comando **appletviewer** proporcionado con el J2SDK; el conjunto de herramientas que incluye el compilador (**javac**), el intérprete (**java**), el **appletviewer** y otras herramientas utilizadas por los programadores de Java. Tal como Netscape Communicator, Internet Explorer y HotJava, el **appletviewer** requiere un documento HTML para invocar un applet. Por ejemplo, si el archivo **Bienvenido.html** hace referencia al applet **Bienvenido**, el comando **appletviewer** se utiliza de la siguiente manera:

```
appletviewer Bienvenido.html
```

Esto provoca que el cargador de clases cargue la información utilizada en el applet **Bienvenido**. Por lo general, al **appletviewer** se le conoce como un “navegador mínimo”; éste sólo sabe cómo interpretar applets.

Antes de que se ejecuten los bytecodes de un applet por medio del interprete de Java, incluido en un navegador o mediante el **appletviewer**, éstos se verifican por medio del *verificador de bytecode* de la fase 4 (esto también sucede con aplicaciones que descargan bytecodes desde una red). Esto garantiza que los bytecodes para la clases que se cargan desde Internet (conocidas como *clases descargables*) sean válidas y que no violen las restricciones de seguridad de Java. Java promueve una fuerte seguridad debido a que los programas en Java que llegan desde una red no deben ser capaces de dañar a sus archivos y a su sistema (como lo hacen los virus).

Por último, en la fase 5, la computadora, bajo el control de su CPU, interpreta el programa un bytecode a la vez, y realiza las acciones especificadas en el programa.

Es posible que los programas no funcionen en el primer intento. Cada una de las fases anteriores puede fallar debido a los diversos errores que explicaremos en este libro. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal en Java, como en la aritmética). Esto provocaría que el programa en Java imprimiera un mensaje de error. El programador regresaría a la fase de edición, haría las correcciones necesarias y procedería de nuevo a través de las fases restantes, para determinar si las correcciones funcionan correctamente.

Error común de programación 24.1



Los errores como la división entre cero ocurren durante la ejecución del programa, de modo que estos errores se llaman errores en tiempo de ejecución o errores de ejecución. Los errores fatales en tiempo de ejecución provocan que los programas terminen de inmediato, sin tener éxito al realizar sus tareas. Los errores no fatales en tiempo de ejecución permiten a los programas completar su ejecución, por lo general con resultados incorrectos.

La mayoría de los programas en Java introducen datos de entrada y/o salida. Cuando decimos que un programa imprime un resultado, por lo general significa que el resultado se despliega en la pantalla. Los datos pueden emitirse con otros dispositivos tales como discos e impresoras.

24.3 Notas generales acerca de Java y de este libro

Java es un lenguaje poderoso. Algunas veces, los programadores experimentados se enorgullecen de su capacidad para usar el lenguaje de manera extraña, contorsionada, y compleja. Ésta es una pobre práctica de programación. Hace que los programas sean más difíciles de leer, más propensos a comportarse de manera extraña, más difíciles de depurar y probar, y más difíciles de adaptar a los requerimientos cambiantes. Estos capítulos también están dedicados a los programadores novatos, de modo que anteponemos la *claridad*. Ésta es nuestra primera “buena práctica de programación”.



Buena práctica de programación 24.1

Escriba sus programas en Java de manera sencilla y directa. A esto en ocasiones se le llama KIS (“keep it simple”, “manténgalo simple”). No deshaga el lenguaje, intentando usos extraños.

Usted habrá escuchado que Java es un lenguaje portable, y que los programas escritos en Java pueden ejecutarse en muchas computadoras diferentes. *La portabilidad es una meta escurridiza*. El documento del estándar de C contiene una larga lista de temas de portabilidad, y se han escrito libros completos para explicarla.



Tip de portabilidad 24.1

Aunque es más fácil escribir programas portables en Java que en la mayoría de los demás lenguajes de programación, existen diferencias entre los compiladores, los intérpretes y las computadoras que pueden hacer de la portabilidad una meta difícil de alcanzar. El simple hecho de escribir programas en Java, no garantiza la portabilidad. Ocasionalmente el programador necesitará lidiar directamente con las variaciones entre los compiladores y las computadoras.



Tip para prevenir errores 24.1

Siempre pruebe los programas en Java en todos los sistemas en los que desee ejecutarlos.

Hicimos un cuidadoso recorrido a través de la documentación de Java de Sun, y comparamos nuestra programación con ésta por motivos de integridad y exactitud. Sin embargo, Java es un lenguaje rico, y existen algunas sutilezas en el lenguaje y algunos temas que no hemos cubierto. Si usted necesita detalles técnicos adicionales, le sugerimos que lea el documento más reciente de Java disponible en Internet y en java.sun.com.

Buena práctica de programación 24.2



Lea la documentación para la versión de Java que va a utilizar. Consulte esta documentación con frecuencia para asegurarse de que conoce la rica colección de características de Java y de que utiliza correctamente estas características.

Buena práctica de programación 24.3



Su computadora y su compilador son buenos maestros. Si después de leer cuidadosamente el manual de la documentación de Java no está seguro de la manera en que funciona una característica de Java, experimente y vea qué sucede. Estudie cada mensaje de error o de advertencia que obtenga cuando compile sus programas, y corríjalos para eliminar dichos mensajes.

Aquí explicamos cómo funciona Java en su implementación común. Quizá el problema más grave con las primeras versiones de Java es que los programas en Java se ejecutan mediante un intérprete en la máquina del cliente. Los intérpretes se ejecutan muy lento, comparados con los programas totalmente compilados en lenguaje máquina.

Tip de rendimiento 24.1



Los intérpretes tienen una ventaja sobre los compiladores en el mundo de Java, a saber, que un programa interpretado puede comenzar su ejecución de inmediato, tan pronto como se descarga en la máquina del cliente, mientras que un programa a compilarse primero debe sufrir un retraso potencialmente largo mientras el programa se compila antes de que pueda ejecutarse.

Aunque en los primeros sistemas Java solamente los intérpretes estaban disponibles para ejecutar los bytecodes en el sitio del cliente, los compiladores de Java se escribieron para la mayoría de las plataformas más populares. Estos compiladores toman los bytecodes de Java (o en algunos casos el código fuente de Java) y los compilan en el código de máquina nativo de la máquina del cliente. Estos programas compilados se desempeñan de manera similar al código compilado de C o C++. No existen compiladores para cada plataforma Java, de modo que los programas no podrán ejecutarse al mismo nivel en todas las plataformas.

Los applets presentan algunas características más interesantes. Recuerde, un applet podría provenir virtualmente desde cualquier *servidor Web* del mundo. De modo que el applet tendrá que ser capaz de ejecutarse en cualquier plataforma de Java. En resumen, los applets de rápida ejecución de Java realmente pueden interpretarse. Pero, ¿qué sucede con los applets más grandes y de cómputo intensivo? Aquí, el usuario podría estar dispuesto a sufrir el retraso de la compilación para obtener un mejor rendimiento de ejecución. Para algunos applets especializados de alto rendimiento, el usuario pudiera no tener opción; el código interpretado se ejecutaría muy lentamente para que el código del applet se ejecutara apropiadamente, por lo que el applet tendría que compilarse.

Un paso intermedio entre los intérpretes y los compiladores es un *compilador justo a tiempo (JIT, just in time)* que, mientras se ejecuta el compilador, produce código compilado para los programas y los ejecuta en lenguaje máquina, en lugar de reinterpretarlos. Los compiladores JIT no producen código máquina, que es tan eficiente como un compilador completo. En la actualidad, los compiladores completos para Java se encuentran en desarrollo. Para obtener la información más reciente sobre la traducción de un programa en Java a alta velocidad, puede leer acerca del compilador *HotSpot* de Sun, visite

java.sun.com/products/spot/

Para las empresas que quieren desarrollar sistemas de información de trabajo pesado, los ambientes integrados de desarrollo (IDEs) de las empresas de software más importantes están disponibles. Los IDEs proporcionan muchas herramientas para soportar el proceso de desarrollo de software. Actualmente, muchos IDEs de Java en el mercado son tan poderosos como aquellos disponibles para el desarrollo de sistemas en C y C++. Ésta es una clara señal de que Java ya ha sido aceptado como un lenguaje viable para el desarrollo de importantes sistemas de software.

24.4 Un programa sencillo: Impresión de una línea de texto

Comenzaremos considerando una sencilla *aplicación* en Java que despliega una línea de texto. Una aplicación es un programa que se ejecuta por medio del intérprete **java** (el cual explicaremos más adelante en esta sección). El programa y su salida aparecen en la figura 24.2.

Este programa muestra varias características importantes del lenguaje Java. Consideraremos con detalle cada línea del programa. Cada programa tiene las líneas numeradas, para conveniencia del lector; dichos números de línea no son parte de los programas Java. La línea 7 hace el “trabajo real” del programa, a saber, despliega en la pantalla la frase **Bienvenido a la programacion en Java!**. Pero, consideremos cada línea en orden. La línea 1

```
// Figura 24.2: Bienvenido1.java
```

comienza con `//`, lo que indica que el resto de la línea es un *comentario*. Comenzamos cada programa con un comentario que indica el número y el nombre del archivo. Como en C++, a un comentario que comienza con `//` se le llama *comentario de una sola línea*, debido a que el comentario termina al final de la línea actual.

Java también soporta comentarios de varias líneas (delimitados con `/*` y `*/`), los cuales presentamos en el capítulo 2; una manera similar de hacer comentarios, llamada *comentario para documentación*, se delimita con `/**` y `*/`.



Error común de programación 24.2

Olvidar uno de los delimitadores de un comentario de varias líneas, es un error de sintaxis.

Por lo general, los programadores en Java utilizan comentarios de una sola línea al estilo C++, con más preferencia que los comentarios al estilo C. A través de este libro, utilizaremos comentarios de una sola línea al estilo C++. Java introdujo la sintaxis del comentario de documentación para permitir a los programadores resaltar porciones de programas, que el programa de utilidad **javadoc** (proporcionado por Sun Microsystems con el Java 2 Software Development Kit) pueda leer y utilizar para preparar automáticamente la documentación de sus sistemas. Existen algunos aspectos sutiles para utilizar adecuadamente los comentarios estilo **javadoc** dentro de un programa. En este libro no utilizaremos los comentarios al estilo **javadoc**.

La línea 4

```
public class Bienvenido1 {
```

comienza la *definición de la clase* **Bienvenido1**. Cada programa en Java consta de al menos una definición de clase definida por usted, el programador. A estas clases se les conoce como *clases definidas por el programador* o *clases definidas por el usuario*. En el capítulo 26, explicamos programas que contienen varias clases definidas por el programador. La *palabra reservada* **class** introduce la definición de una clase en Java y va inmediatamente seguida por el *nombre de la clase* (**Bienvenido1** en este programa). Las palabras reservadas (o palabras clave) se reservan para el uso de Java (a lo largo del libro explicamos las palabras reservadas), y siempre se escriben con letras minúsculas. Por convención, todos los nombres de las clases en Java comienzan

```

1 // Figura 24.2: Bienvenido1.java
2 // Primer programa en Java
3
4 public class Bienvenido1 {
5     public static void main( String args[] )
6     {
7         System.out.println( "Bienvenido a la programacion en Java!" );
8     } // fin de main
9 } // fin de la clase Bienvenido1

```

Bienvenido a la programacion en Java!

Figura 24.2 Primer programa en Java.

con una letra mayúscula, y tienen una letra mayúscula por cada palabra en el nombre de la clase (por ejemplo, **NombreClaseEjemplo**). Al nombre de la clase se le llama *identificador*. Un identificador es una serie de caracteres que consta de letras, dígitos, guiones bajos (_) y símbolos de moneda (\$), que no comienzan con un dígito y no contienen espacios. Algunos identificadores válidos son **Bienvenido1**, **\$valor**, **_valor**, **m_campoEntrada1**, y **boton7**. El nombre **7boton** no es un identificador válido debido a que comienza con un dígito, y el nombre **campo entrada** no es un identificador válido debido a que contiene un espacio. Java es *sensible a mayúsculas y minúsculas*, las letras mayúsculas y minúsculas son diferentes, de modo que **a1** y **A1** son identificadores diferentes.

Error común de programación 24.3



Java es sensible a mayúsculas y minúsculas. Por lo general, no utilizar las letras mayúsculas y minúsculas apropiadas para un identificador, es un error de sintaxis.

Buena práctica de programación 24.4



Por convención, usted siempre debe comenzar el nombre de una clase con la primera letra en mayúscula.

Buena práctica de programación 24.5



Cuando lea un programa en Java, busque identificadores que comiencen con la primera letra en mayúscula. Por lo general, éstos representan clases de Java.

Observación de ingeniería de software 24.1



Evite utilizar identificadores que contengan signos de moneda (\$), ya que con frecuencia el compilador los utiliza para crear nombres de identificadores.

En los capítulos 24 y 25, toda clase que definimos comienza con la *palabra reservada public*. Por ahora, solamente requeriremos esta palabra reservada. En el capítulo 26, explicaremos con detalle la palabra reservada **public**, y también explicaremos las clases que no comienzan con dicha palabra reservada. [Nota: En este libro, muchas veces le pedimos que simplemente imitara ciertas características de Java que presentábamos mientras usted escribía sus propios programas en Java. Esto lo hacemos específicamente cuando aún no es importante conocer todos los detalles acerca de una característica de Java. De inicio, todos los programadores aprenden cómo programar, imitando lo que otros programadores han hecho antes que ellos. En cada detalle que le pedimos que imite, le indicamos en dónde se encuentra la explicación completa que le daremos más adelante.]

Cuando usted guarda la definición de una clase dentro de un archivo, el nombre de la clase debe utilizarse como parte del nombre del archivo. Para nuestras aplicaciones, el nombre del archivo es **Bienvenido1.java**. Todas las definiciones de clases en Java se almacenan en archivos que terminan con la extensión de archivo **.java**.

Error común de programación 24.4



Para una clase pública, es un error si el nombre de archivo no es idéntico al nombre de la clase tanto en las letras, como en las mayúsculas y las minúsculas. Por lo tanto, también es un error que un archivo contenga dos o más clases públicas.

Error común de programación 24.5



Es un error no finalizar el nombre de un archivo con la extensión **.java**, si contiene la definición una clase de la aplicación. El compilador de Java no podrá compilar la definición de la clase.

Una *llave izquierda* (al final de la línea 4 de este programa), {, comienza el *cuerpo* de cada definición de clase. Observe que las líneas 5 a 8 están sangradas. Ésta es una convención de espaciado utilizada para hacer más legibles los programas. Definimos cada convención de espaciado como una *buena práctica de programación*.

Error común de programación 24.6



Si las llaves no están en pares coincidentes, el compilador indica un error.



Buena práctica de programación 24.6

Cada vez que introduzca una llave izquierda de apertura, {, en su programa, introduzca inmediatamente la llave derecha de cierre, }, y vuelva a colocar el indicador entre las llaves para comenzar a introducir el cuerpo del programa. Esto ayuda a evitar que falten llaves.



Buena práctica de programación 24.7

Sangre el cuerpo entero de cada definición de clase un “nivel” entre la llave izquierda, {, y la llave derecha, }, que define el cuerpo de la clase. Esto enfatiza la estructura de la definición de la clase, y ayuda a que las definiciones de clases sean más fáciles de leer.



Buena práctica de programación 24.8

Establezca una convención para el tamaño del sangrado que prefiera, y entonces aplique de manera uniforme dicha convención. Puede utilizar la tecla tab para crear el sangrado, aunque tab podría variar entre editores. Le recomendamos el uso de tabuladores de 1/4 de pulgada o (preferiblemente) tres espacios para formar un nivel de sangrado.

La línea 5

```
public static void main( String args[] )
```

es parte de cada aplicación en Java. Las aplicaciones en Java comienzan automáticamente en **main**. Los paréntesis después de **main** indican que **main** es un *método* de programa, o lo que un programador en C o C++ llamaría una función. Por lo general, las definiciones de clases en Java contienen uno o más métodos. Para una clase de aplicación en Java, exactamente uno de esos métodos debe llamarse **main** y debe definirse como muestra la línea 5; de lo contrario, el intérprete de java no ejecutará la aplicación. Los métodos son capaces de realizar tareas y devolver información cuando llevan a cabo sus funciones. La palabra reservada **void** indica que este método realizará una tarea (en este programa despliega una línea de texto), pero no devolverá información alguna cuando complete su tarea. Veremos que muchos métodos devuelven información cuando completan su tarea. En el capítulo 25 explicaremos con detalle los métodos. Por ahora, simplemente imite la primera línea en cada una de las aplicaciones en Java.

La llave izquierda, {, de la línea 6 comienza el *cuerpo de la definición del método*. Su correspondiente llave derecha, }, debe terminar el cuerpo de la definición del método (línea 8 del programa). Observe que la línea en el cuerpo del método se sangra entre las dos llaves.



Buena práctica de programación 24.9

Sangre por completo el cuerpo de cada definición de método un “nivel” entre la llave izquierda, {, y la llave derecha, }. Esto hace que la estructura del método resalte, y ayuda a que la definición del método sea más fácil de leer.

La línea 7

```
System.out.println( "Bienvenido a la programacion en Java!" );
```

instruye a la computadora para que imprima la *cadena* de caracteres contenida entre las comillas dobles. En ocasiones, a una cadena se le llama *cadena de caracteres*, un *mensaje* o una *literal de cadena*. Por lo general, nos referiremos a los caracteres entre comillas como cadenas. El compilador no ignora los caracteres blancos en una cadena.

A **System.out** se le conoce como *objeto estándar de salida*. **System.out** permite a las aplicaciones en Java desplegar cadenas y otro tipo de información en la *ventana de comando* desde la que se ejecuta la aplicación en Java. En Windows 95/98 de Microsoft, la ventana de comando es el *indicador de MS-DOS*. En Windows NT de Microsoft, la ventana de comando es el *Indicador de comandos*. En UNIX, a la ventana de comando por lo general se le llama *ventana de comando, herramienta shell o shell*. En computadoras que ejecutan un sistema operativo que no tiene una ventana de comando (tal como Macintosh), el intérprete **java** por lo general despliega una ventana que contiene la información que despliega el programa.

El método **System.out.println** despliega (o imprime) una *línea* de texto en la ventana de comando. Cuando **System.out.println** completa su tarea, coloca el *cursor de salida* (la ubicación en donde se desplegará el siguiente carácter) al principio de la siguiente línea en la ventana de comando (esto es similar a oprir

mir la tecla *Entrar*, cuando escribe en un editor de texto; el cursor se reposiciona al principio de la siguiente línea de su archivo).

A la línea completa, incluido **System.out.println**, su argumento en los paréntesis (la cadena) y el punto y coma (;), se le llama *instrucción*. Toda instrucción debe terminar con un punto y coma (también llamado *terminador de instrucción*). Cuando se ejecuta esta instrucción, se despliega el mensaje **Bienvenido a la programacion en Java!** en la ventana de comando.



Error común de programación 24.7

Omitir el punto y coma al final de una instrucción, es un error de sintaxis.



Tip para prevenir errores 24.2

Cuando el compilador reporta un error de sintaxis, el error podría no estar en la línea que indica el mensaje de error. Primero, verifique la línea en donde se reporta el error. Si la línea no contiene errores de sintaxis, verifique las líneas anteriores del programa.

Ahora estamos listos para compilar y ejecutar nuestro programa. Para compilar el programa, abrimos la ventana de comando, nos cambiamos al directorio en donde se encuentra almacenado el programa y escribimos

```
javac Bienvenido1.java
```

Si el programa no contiene errores de sintaxis, el comando anterior crea un nuevo archivo llamado **Bienvenido1.class** que contiene los bytecodes de Java que representan a nuestra aplicación. Estos bytecodes serán interpretados por el intérprete **java** cuando le indiquemos que ejecute el programa al escribir el comando

```
java Bienvenido1
```

el cual inicia el intérprete e indica que debe cargarse el archivo **.class** para la clase **Bienvenido1**. Obsérve que se omite la extensión **.class** del nombre del archivo del comando anterior; de lo contrario, el intérprete no ejecutará el programa. El intérprete llama automáticamente al método **main**. A continuación, la instrucción de la línea 7 de **main** despliega “**Bienvenido a la programacion en Java!**”. La figura 24.3 muestra la ejecución de la aplicación en la ventana de MS-DOS.

Aunque este primer programa despliega la salida en la ventana de comandos, la mayoría de las aplicaciones Java que despliegan la salida utilizan ventanas o *cuadros de diálogo*. Por ejemplo, los navegadores de la World Wide Web, tales como Netscape Communicator o Microsoft Internet Explorer despliegan las páginas Web en sus propias ventanas. Por lo general, los programas de correo electrónico le permiten escribir un mensaje en una ventana proporcionada por el programa de correo electrónico, o leer los mensajes que recibe en una ventana proporcionada por el programa de correo electrónico. Los cuadros de diálogo son ventanas que por lo general se utilizan para desplegar mensajes importantes para el usuario de una aplicación. Java 2 ya incluye la clase **JOptionPane** que le permiten desplegar fácilmente un cuadro de diálogo que contiene información. El programa de la figura 24.4 despliega una cadena similar a la que aparece en la figura 24.2 en un cuadro de diálogo predefinido llamado *diálogo de mensaje*. Observe que esta nueva versión del programa también utiliza la secuencia de escape al estilo C, \n, para insertar en la cadena caracteres de nueva línea.

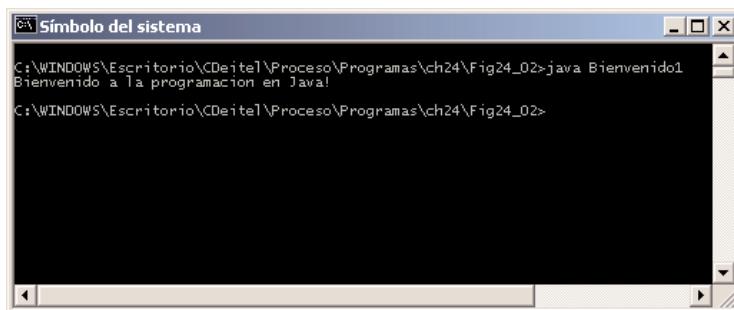


Figura 24.3 Ejecución de la aplicación **Bienvenido1** en la ventana de MS-DOS (Símbolo del sistema).

```

1 // Figura 24.4: Bienvenido2.java
2 // Impresión de múltiples líneas en un cuadro de diálogo
3 import javax.swing.JOptionPane; // importa la clase JOptionPane
4
5 public class Bienvenido2 {
6     public static void main( String args[ ] )
7     {
8         JOptionPane.showMessageDialog(
9             null, "Bienvenido\na la\nprogramacion\nen Java!" );
10
11     System.exit( 0 ); // termina el programa
12 } // fin de main
13 } // fin de la clase Bienvenido2

```



Figura 24.4 Cómo desplegar varias líneas en un cuadro de diálogo.

Una de las grandes fortalezas de Java es su rica colección de clases predefinidas, las cuales pueden reutilizar los programadores en lugar de “reinventar la rueda”. En el libro, utilizamos un gran número de estas clases. Las diversas clases predefinidas se agrupan en categorías de clases relacionadas llamadas *paquetes*. A los paquetes se les conoce de manera colectiva como *biblioteca de clases de Java* o como la *interfaz de programación de aplicaciones de Java (API)*. La clase **JOptionPane** está definida para nosotros en un paquete llamado **javax.swing**.

La línea 3

```
import javax.swing.JOptionPane;
```

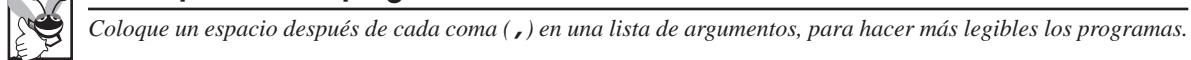
es una instrucción para *importar*. El compilador utiliza instrucciones **import** para identificar y cargar las clases requeridas para compilar un programa en Java. Cuando usted utiliza clases de la API de Java, el compilador intenta garantizar que usted las utiliza correctamente. Las instrucciones **import** ayudan al compilador a localizar las clases que intenta utilizar. Cada porción del nombre del paquete es un directorio (o carpeta) en el disco. Todos los paquetes en la API de Java se almacenan en el directorio **java** o **javax** que contiene muchos subdirectorios, incluso **swing** (un subdirectorio de **javax**). En el capítulo 26, explicaremos con detalle los paquetes.

La línea anterior le indica al compilador que cargue la clase **JOptionPane** desde el paquete **javax.swing**. Este paquete contiene muchas clases que ayudan a los programadores en Java a definir un interfaces gráficas de usuario (GUI) para sus aplicaciones. Los *componentes GUI* facilitan la entrada de datos por parte del usuario de su programa, y el dar formato o presentar la salida de datos para el usuario de su programa. Por ejemplo, la figura 24.5 contiene una ventana de Microsoft Internet Explorer. En la ventana, existe una barra que contiene *menús* (**Archivo**, **Edición**, **Ver**, etcétera). Bajo la barra de menú hay un conjunto de *botones*, los cuales tienen una tarea definida dentro del Microsoft Internet Explorer. Debajo de los botones existe un *campo de texto* en el que el usuario puede introducir el nombre del sitio a visitar dentro de la World Wide Web. A la izquierda del campo de texto existe una *etiqueta* que indica el propósito del campo de texto. Los menús, botones, campos de texto y etiquetas forman parte del GUI del Microsoft Internet Explorer. Todos ellos le permiten a usted interactuar con el programa Explorer. Java contiene clases que implementan los componentes de la GUI descritas aquí, y otras que describiremos en el capítulo 29. En **main**, las líneas 8 y 9

```
JOptionPane.showMessageDialog(
    null, "Bienvenido\na la programacion\nen Java!" );
```

indican una llamada al método `showMessageDialog` de la clase `JOptionPane`. El método requiere dos argumentos. Cuando un método requiere varios argumentos, éstos se separan con *comas* (,). Hasta que expliquemos `JOptionPane` con detalle en el capítulo 29, el primer argumento siempre será la palabra reservada `null`. El segundo argumento es la cadena a desplegar.

Buena práctica de programación 24.10



El método `JOptionPane.showMessageDialog` es un método de la clase `JOptionPane` llamado *método estático*. Dichos métodos siempre se llaman mediante el nombre de la clase, seguido por un operador punto (.) y el nombre del método. En el capítulo 26, explicaremos los métodos estáticos.

Al ejecutar la instrucción anterior se despliega el cuadro de diálogo que muestra la figura 24.6. La *barra de título* del diálogo contiene la cadena **Message** para indicar que el diálogo presenta un mensaje para el usuario. El cuadro de diálogo incluye automáticamente el botón **ACEPTAR** que permite al usuario utilizarlo para *retirar (ocultar) el diálogo* al presionar el botón. Esto se lleva a cabo colocando el *cursor del ratón* (también llamado *apuntador del ratón*) sobre el botón **ACEPTAR** y haciendo clic con el ratón.

Recuerde que todas las instrucciones en Java terminan con un punto y coma (;). Por lo tanto, las líneas 8 y 9 representan una instrucción. Java permite a instrucciones grandes dividirse en varias líneas. Sin embargo, usted no puede dividir una instrucción en medio de un identificador o en el centro de la cadena.

Error común de programación 24.8

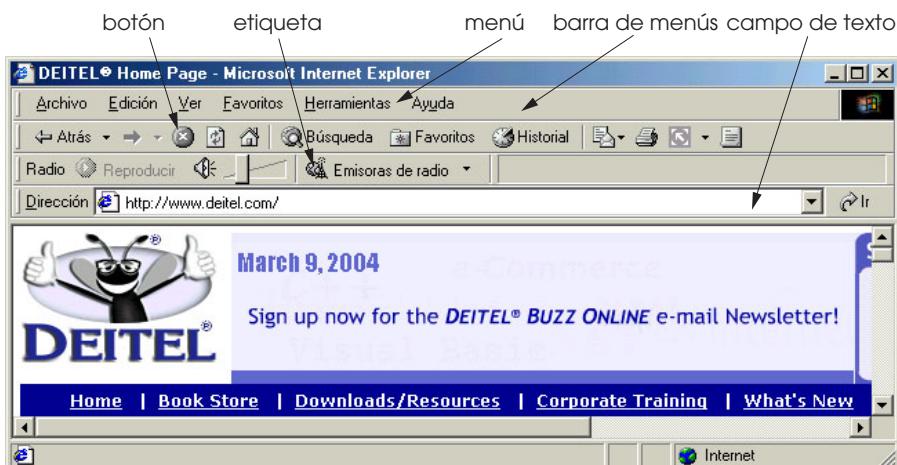
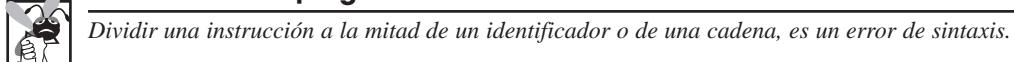


Figura 24.5 Ventana de Microsoft Internet Explorer con componentes GUI.

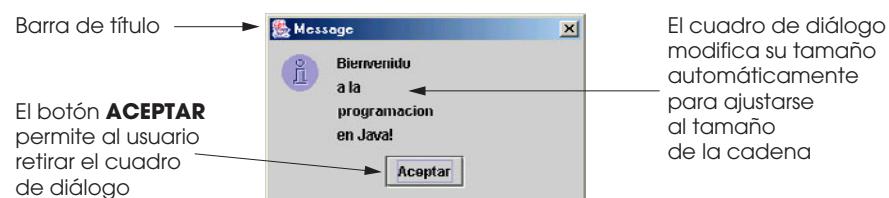


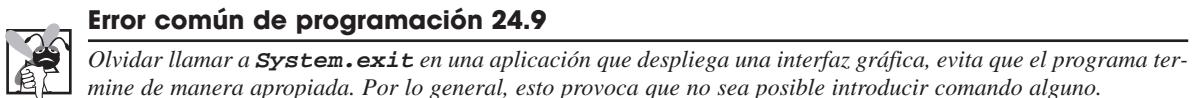
Figura 24.6 Diálogo de mensaje.

La línea 11

```
System.exit( 0 ); // termina el programa
```

utiliza el método estático **exit** de la clase **System** para terminar la aplicación. Esta línea es necesaria en cualquier aplicación que despliegue una interfaz gráfica de usuario, para terminar la aplicación. De nuevo, observe la sintaxis utilizada para llamar al método, el nombre de la clase (**System**), un punto (.) y un nombre de método (**exit**). Recuerde que los identificadores que comienzan con una letra mayúscula, por lo general representan nombres de clases. Por lo tanto, usted puede asumir que **System** es una clase. El argumento **0** para el método **exit** indica que las aplicaciones terminaron exitosamente (por lo general un valor diferente de cero indica que ocurrió un error). Este valor se pasa a la ventana de comando que ejecuta el programa. Esto es útil si el programa se ejecuta desde un archivo batch (en sistemas Windows 95/98/NT), o mediante un script del shell (en sistemas UNIX). Por lo general, los archivos batch y los scripts se utilizan para ejecutar programas en secuencia, de manera que cuando termina el primer programa, comienza automáticamente la ejecución del siguiente programa. Para mayor información acerca de los archivos batch o los scripts del shell, revise la documentación de su sistema operativo.

La clase **System** es parte del paquete **java.lang**. Observe que la clase **System** no se importa mediante una instrucción **import** al principio del programa. El paquete **java.lang** se importa automáticamente en cada programa en Java.



24.5 Otra aplicación en Java: Suma de enteros

Nuestra siguiente aplicación introduce dos enteros (números completos) escritos por el usuario desde el teclado, calcula la suma de estos valores y despliega el resultado. Conforme el usuario introduce cada entero y presiona la tecla *Entrar*, el entero se introduce en el programa y se suma al total.

Este programa utiliza otro cuadro de diálogo predefinido desde la clase **JOptionPane** llamado *diálogo de entrada*, el cual permite al usuario introducir un valor a utilizarse en el programa. El programa también utiliza un diálogo de mensaje para desplegar los resultados de la suma. La figura 24.7 muestra la aplicación y las capturas de las pantallas de prueba.

```

1 // Figura 24.7: Suma.java
2 // Un programa de suma
3
4 import javax.swing.JOptionPane; // importa la clase JOptionPane
5
6 public class Suma {
7     public static void main( String args[] )
8     {
9         String primerNumero,           // primera cadena introducida por el usuario
10            segundoNumero;          // segunda cadena introducida por el usuario
11         int numero1,                // primer número a sumar
12            numero2,                // segundo número a sumar
13            suma;                  // suma de numero1 y numero2
14
15         // lee el primer número del usuario como una cadena
16         primerNumero =
17             JOptionPane.showInputDialog( "Introduzca el primer entero" );
18
19         // lee el segundo número del usuario como una cadena

```

Figura 24.7 Un programa de suma “en acción”. (Parte 1 de 2.)

```

20     segundoNumero =
21         JOptionPane.showInputDialog( "Introduzca el segundo entero" );
22
23     // convierte los números del tipo String a tipo int
24     numerol = Integer.parseInt( primerNumero );
25     numero2 = Integer.parseInt( segundoNumero );
26
27     // suma los números
28     suma = numerol + numero2;
29
30     // despliega los resultados
31     JOptionPane.showMessageDialog(
32         null, "La suma es " + suma, "Resultados",
33         JOptionPane.PLAIN_MESSAGE );
34
35     System.exit( 0 );    // termina el programa
36 } // end main
37 } // fin de la clase Suma

```

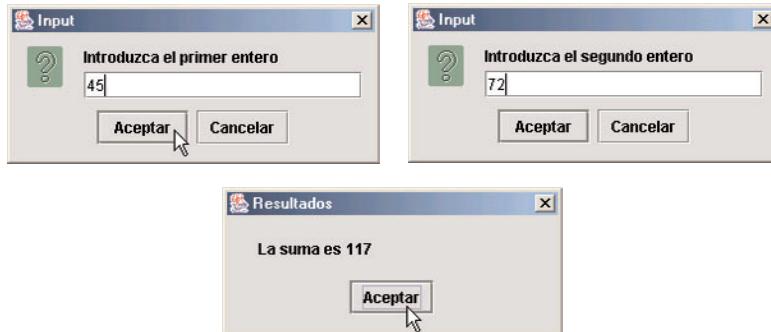


Figura 24.7 Un programa de suma “en acción”. (Parte 2 de 2.)

La línea 4

```
import javax.swing.JOptionPane;    // importa la clase JOptionPane
```

especifica al compilador en dónde localizar **JOptionPane** para utilizarlo con esta aplicación.

Como lo establecimos previamente, todo programa en Java consta de al menos una definición de clase. La línea 6

```
public class Suma {
```

comienza las definiciones de la clase **Suma**. El nombre de archivo para esta clase pública debe ser **Suma.java**.

Recuerde que todas las definiciones de clases comienzan con una llave izquierda de apertura (final de la línea 6), {, y con una llave derecha de cierre, } (línea 37).

Como establecimos anteriormente, toda aplicación comienza su ejecución con el método **main** (línea 7). La llave izquierda (línea 8) marca el inicio del cuerpo de **main** y su correspondiente llave izquierda (línea 36) marca el final.

Las líneas 9 y 10

```
String primerNumero,      // primera cadena introducida por el usuario
       segundoNumero       // segunda cadena introducida por el usuario
```

forman una *declaración*. Las palabras **primerNumero** y **segundoNumero** son los nombres de las *variables*. Todas las variables deben declararse con el nombre y el tipo de dato, antes de que puedan utilizarse dentro de

un programa. Esta declaración especifica que las variables **primerNúmero** y **segundoNúmero** son tipos de datos **String** (del paquete **java.lang**), lo cual significa que estas variables almacenarán cadenas. Un nombre de variable puede ser cualquier identificador válido. Las declaraciones terminan con punto y coma (**;**), y pueden dividirse en varias líneas, con cada variable en la declaración separada por una coma (es decir, una *lista separada por comas* de nombres de variables). Es posible declarar varias variables en una declaración o en declaraciones múltiples. Podríamos haber escrito dos declaraciones, una para cada variable, pero la declaración anterior es más concisa. Observe los comentarios de una sola línea al final de cada línea. Ésta es una sintaxis común utilizada por los programadores para indicar el propósito de cada variable en el programa.



Buena práctica de programación 24.11

Elegir nombres de variables significativas (descriptivas) ayuda a un programa a estar “autodocumentado” (es decir, se vuelve más sencillo comprender un programa sólo con leerlo, y no es necesario tener que leer los manuales o utilizar comentarios en exceso).



Buena práctica de programación 24.12

*Por convención, los identificadores de nombres de variables comienzan con una letra minúscula. Así como con los nombres de las clases, cada palabra del nombre después de la primera, debe comenzar con una letra mayúscula. Por ejemplo, el identificador **primerNúmero** tiene una letra mayúscula **N** en la segunda palabra **Número**.*



Buena práctica de programación 24.13

Algunos programadores prefieren declarar cada variable en una línea aparte. Este formato permite insertar fácilmente un comentario descriptivo después de cada declaración.

En las líneas 11 a 13

```
int numero1,      // primer número a sumar
    numero2,      // segundo número a sumar
    suma;         // suma el numero1 y el numero2
```

declaran que las variables **numero1**, **numero2** y **suma** son datos de tipo **int**.

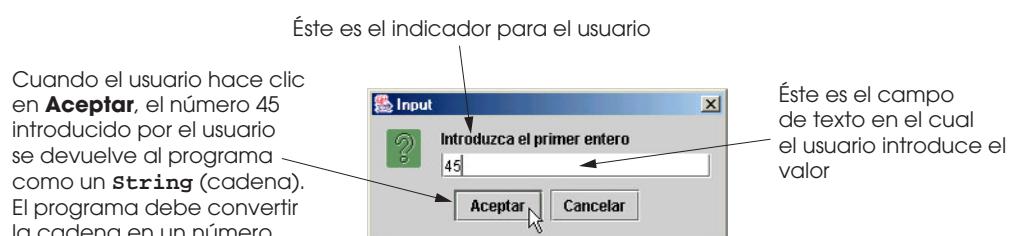
Pronto explicaremos los tipos de datos **float** y **double** para especificar números reales y variables de tipo **char** para especificar datos de tipo carácter. Una variable **char** puede contener solamente una letra minúscula, una letra mayúscula, un solo dígito, o un carácter especial tal como **X**, **\$**, **7**, ***** y secuencias de escape (tales como el carácter de nueva línea **\n**). Además, Java es capaz de representar caracteres de muchos otros idiomas.

Con frecuencia, a los tipos tales como **int**, **double** y **char** se les llama *tipos de datos primitivos* o *tipos de datos predefinidos*. Los nombres de los tipos primitivos son palabras reservadas. En el capítulo 25 resumimos los ocho tipos de datos primitivos (**boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**).

Las líneas 15 a 17

```
// lee el primer número del usuario como una cadena
primerNúmero =
    JOptionPane.showInputDialog ("Introduzca el primer entero" );
```

lee un **String** introducido por el usuario que representa el primero de dos enteros que se sumarán. El método **JOptionPane.showInputDialog** despliega el siguiente cuadro de diálogo:



El argumento de `showInputDialog` indica al usuario qué hacer en el siguiente campo. A este mensaje se le llama *indicador* debido a que le señala al usuario que realice una acción específica. El usuario escribe caracteres en el campo de texto, luego hace clic en el botón **Aceptar** para devolver la cadena al programa. [Si usted escribe y nada aparece en el campo de texto, coloque el apuntador del ratón en el campo de texto y haga clic con el ratón para activar dicho campo.] Desafortunadamente Java no proporciona una forma sencilla de entrada que sea análoga, para desplegar una salida en la ventana de comandos con `System.out.print` y `System.println`. Por esta razón, normalmente recibimos la entrada de un usuario a través de un componente GUI (un diálogo de entrada en este programa).

Técnicamente el usuario puede escribir cualquier cosa en el campo de texto de entrada. En este programa, si el usuario digita un valor no entero o hace clic en el botón **Cancelar**, ocurrirá un error lógico en tiempo de ejecución.

El resultado de llamar a `JOptionPane.showInputDialog` (un `String` que contiene los caracteres digitados por el usuario) se da a la variable `primerNumero` con el operador de asignación `=`. La instrucción se lee como, “`primerNumero` obtiene el valor `JOptionPane.showInputDialog("Introduzca el primer entero")`”. El operador `=` es un operador binario debido a que tiene dos operandos: `primerNumero` y el resultado de la expresión `JOptionPane.showInputDialog("Introduzca el primer entero")`. A esta instrucción completa se le llama *instrucción de asignación*, debido a que asigna un valor a una variable. La expresión al lado derecho del operador de asignación `=`, siempre se evalúa primero.

Las líneas 19 a 21

```
// lee el segundo número del usuario como una cadena
segundoNumero =
    JOptionPane.showInputDialog( "Introduzca el segundo entero" );
```

despliegan un diálogo de entrada en el que el usuario escribe un `String` que representa el segundo de los dos enteros que se sumarán.

Las líneas 23 a 25

```
// convierte los números del tipo String a tipo int
numero1 = Integer.parseInt( primerNumero );
numero2 = Integer.parseInt( segundoNumero );
```

convierten dos cadenas introducidas por el usuario a valores `int` que pueden utilizarse en el cálculo. El método `Integer.parseInt` (un método estático de la clase `Integer`) convierte su argumento de tipo `String` a un entero. La clase `Integer` es parte del paquete `java.lang`. El entero devuelto por `Integer.parseInt` de la línea 24 se asigna a la variable `numero1`. Cualquier referencia subsiguiente a `numero1` en el programa utiliza el mismo valor entero. El entero devuelto por `Integer.parseInt` en la línea 25 se asigna a la variable `numero2`. Cualquier referencia subsiguiente a `numero2` en el programa utiliza el mismo valor entero.

La instrucción de asignación de la línea 28

```
suma = numero1 + numero2;
```

calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma` mediante el uso del operador de asignación `=`. La instrucción se lee como, “`suma` obtiene el valor de `numero1 + numero2`”. La mayoría de los cálculos se realiza en instrucciones de asignación.

Buena práctica de programación 24.14



Coloque espacios de cualquier lado de un operador binario. Esto hace que el operador sobresalga y hace al programa más legible.

Después de realizar los cálculos, en las líneas 31 a 33

```
JOptionPane.showMessageDialog(
    null, "La suma es " + suma, "Resultados",
    JOptionPane.PLAIN_MESSAGE );
```

utilizan el método `JOptionPane.showMessageDialog` para desplegar el resultado de la suma. La expresión

```
"La suma es " + suma
```

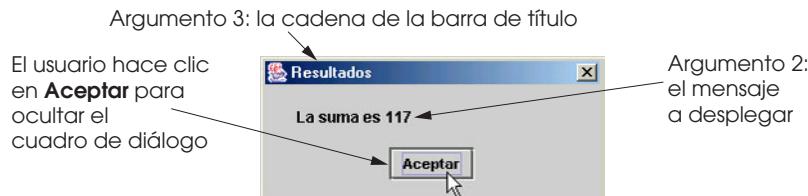
de la instrucción anterior utiliza el operador `+` para “sumar” una cadena (la literal “**La suma es**”) y **suma** (la variable `int` que contiene el resultado de la suma en la línea 28). Java tiene una versión del operador `+` para *concatenación de cadenas* que permite concatenar una cadena y un valor de otro tipo de dato (incluso otra cadena), el resultado de esta operación es una nueva cadena (por lo general más grande). Si asumimos que la suma contiene el valor **117**, la expresión se evalúa de la siguiente manera: Java determina que los dos operandos del operador `+` (la cadena “**La suma es**” y el entero **suma**) son de tipos diferentes y uno de ellos es una cadena. A continuación, **suma** se convierte automáticamente en una cadena y se concatena con “**La suma es**”, lo cual arroja como resultado “**La suma es 117**”. Esta cadena se despliega en el cuadro de diálogo. Observe que la conversión automática de un entero **suma** solamente ocurre debido a que se concatena con la literal de cadena “**La suma es**”. Observe además que el espacio intermedio entre **es** y **117** es parte de la cadena “**La suma es**”.

Error común de programación 24.10



Confundir el operador `+` utilizado para la concatenación de cadenas con el operador `+` utilizado para la suma puede provocar resultados extraños. Por ejemplo, al asumir que la variable entera **y** tiene el valor **5**, la expresión “**y + 2 =**” + **y** + **2** arroja como resultado la cadena “**y + 2 = 52**”, no “**y + 2 = 7**”, debido a que el primer valor de **y** se concatena con la cadena “**y + 2 =**”, después el valor **2** se concatena con la cadena más grande “**y + = 5**”. La expresión “**y + 2 =**” + (**y + 2**) produce el resultado deseado.

La versión del método `showMessageDialog` utilizada en la figura 24.7 es diferente de la que explicamos en la figura 24.4, en la que se requieren cuatro argumentos. El siguiente cuadro de diálogo explica dos de los cuatro argumentos. Así como en la primera versión, el primer argumento siempre será `null` hasta que expliquemos la clase `JOptionPane` con detalle en el capítulo 29. El segundo argumento es el mensaje a desplegar. El tercer argumento es la cadena a desplegar en la barra de título del cuadro de diálogo. El cuarto argumento (`JOptionPane.PLAIN_MESSAGE`) es un valor que indica el tipo de cuadro mensaje a desplegar, este tipo de mensaje no despliega un ícono a la izquierda del mensaje.



En la figura 24.8 mostramos los tipos de diálogos de mensaje. Todos los tipos de diálogo de mensaje, excepto `PLAIN_MESSAGE`, despliegan un ícono para el usuario que indica el tipo de mensaje.

Tipo de diálogo de mensaje	Icono	Descripción
<code>JOptionPane.ERROR_MESSAGE</code>		Despliega un diálogo que indica un error en la aplicación del usuario.
<code>JOptionPane.INFORMATION_MESSAGE</code>		Despliega un diálogo con un mensaje de información sobre la aplicación para el usuario; el usuario simplemente puede ignorar el diálogo.
<code>JOptionPane.WARNING_MESSAGE</code>		Despliega un diálogo que advierte al usuario de la aplicación acerca de un problema potencial.
<code>JOptionPane.QUESTION_MESSAGE</code>		Despliega un diálogo que coloca una pregunta para el usuario de la aplicación. Por lo general, requiere una respuesta tal como hacer clic en el botón Sí o No .
<code>JOptionPane.PLAIN_MESSAGE</code>	sin ícono	Despliega un diálogo que simplemente contiene un mensaje sin ícono.

Figura 24.8 Constantes de `JOptionPane` para los diálogos de mensaje.

24.6 Applets de ejemplo del Java 2 Software Development Kit

Ahora veamos otro tipo de programa en Java: los applets. Comenzaremos nuestra introducción a los applets de Java considerando varios ejemplos proporcionados dentro del Java 2 Software Development Kit (J2SDK) versión 1.4. El applet demuestra una pequeña porción de las poderosas capacidades de Java. Cada programa de ejemplo del J2SDK viene también con el *código fuente* en Java; los archivos **.java** que contienen los applets de Java. Este código fuente será útil mientras progresas en el conocimiento de Java; puede leer el código fuente proporcionado para aprender nuevas y excitantes características de Java. Recuerde, de inicio todos los programadores aprenden nuevos conceptos de programación al imitar el uso de esos conceptos dentro de programas existentes. El J2SDK viene con muchos de esos programas y existe un enorme número de recursos de Java en Internet a través de la World Wide Web que incluyen el código fuente en Java.

Los programas de demostración proporcionados con el J2SDK se localizan en el directorio de instalación de J2SDK dentro de un subdirectorio llamado **demo**. Para el Java 2 Software Development Kit versión 1.4, la ubicación predeterminada para el directorio **demo** en Windows es:

```
c:\j2sdk1.4.1\demo
```

En UNIX/Linux/Mac OS X, es el directorio en el que instala el J2SDK seguido por **j2sdk1.4.1/demo**, por ejemplo,

```
/usr/local/j2sdk1.4.1/demo
```

Para otras plataformas, debe haber una estructura de directorios (o carpetas) similar. En este capítulo asumimos que la J2SDK se instala en **c:\j2sdk1.4.1** en Windows y en su directorio inicial (home) **~/j2sdk1.4.1** en UNIX/Linux/MAC OS X.¹

Si usted utiliza la herramienta de desarrollo de Java que no contiene los demos de Java, puede descargar el J2SDK (con demos) desde el sitio Web de Sun Microsystems

```
java.sun.com/j2se/1.4.1/
```

24.6.1 El applet TicTacToe

El primer applet que demostramos a partir de los demos del J2SDK es el applet llamado **TicTacToe**, el cual nos permite jugar Tic-Tac-Toe (Gato) en contra de la computadora. Para ejecutar este applet, abra una ventana de comando (el indicador de MS-DOS en Windows 95/98/ME, el símbolo del sistema en Windows NT/2000/XP, o una ventana de terminal en UNIX/Linux/Mac OS X) y cambie de directorio hacia el directorio correspondiente al demo del J2SDK. Cada sistema operativo que mencionamos aquí utiliza el comando **cd** para *cambiar de directorio*. Por ejemplo,

```
cd c:\j2sdk1.4.1\demo
```

cambia el directorio activo hacia **demo** en Windows y

```
cd ~/j2sdk1.4.1/demo
```

cambia el directorio activo hacia **demo** en UNIX/Linux/Mac OS X.

El directorio **demo** contiene varios subdirectorios. Usted puede listar estos directorios al escribir **dir** en la ventana de comandos de Windows, o el comando **ls** en UNIX/Linux/Mac OS X. Explicaremos los directorios **applets** y **jfc**. El directorio **applets** contiene muchos applets de demostración. El directorio **jfc** (Java Foundation Classes) contiene muchos ejemplos de las características de gráficos y GUI de Java (algunos de estos ejemplos también son applets). Cambie el directorio activo al directorio applet mediante el siguiente comando

```
cd applets
```

ya sea en Windows o en UNIX/Linux/Mac OS X.

Liste el contenido del directorio **applets** para ver los nombres de directorio para los applets de demostración. La figura 24.9 proporciona una breve descripción de cada ejemplo.

1. Es posible que necesite actualizar estas ubicaciones para reflejar el directorio de instalación que eligió y la unidad de disco, o una versión diferente de J2SDK.

Ejemplo	Descripción
Animator	Ejecuta una de cuatro animaciones diferentes.
ArcTest	Demuestra el dibujo de arcos. Usted puede interactuar con el applet para modificar los atributos del arco que se despliega.
BarChart	Dibuja un gráfico sencillo de barras.
Blink	Despliega texto intermitente en diferentes colores.
CardTest	Demuestra varios componentes GUI y una variedad de formas en las que pueden organizarse los componentes GUI en la pantalla. (La organización de los componentes GUI también se conoce como <i>diseño</i> de los componentes GUI.)
Clock	Dibuja un reloj de manecillas “rotantes”, la fecha y la hora actuales. El reloj se actualiza una vez por segundo.
DitherTest	Demuestra el dibujo con la técnica de gráficos conocida como <i>tramado</i> , la cual permite la transformación gradual de un color a otro.
DrawTest	Permite al usuario arrastrar el ratón para dibujar líneas y puntos de diferentes colores en el applet.
Fractal	Dibuja un fractal. Por lo general, los fractales requieren cálculos complejos para determinar la manera en que se despliegan.
GraphicsTest	Dibuja una variedad de figuras para ilustrar las capacidades gráficas de Java.
GraphLayout	Dibuja un grafo que consta de muchos nodos (representados como rectángulos) conectados mediante líneas. Arrastre un nodo para que vea cómo los demás nodos se ajustan en la pantalla y para demostrar sus complejas interacciones gráficas.
ImageMap	Demuestra una imagen con <i>puntos activos</i> . Al colocar el apuntador del ratón sobre ciertas áreas de la imagen resalta el área y se despliega un mensaje en la esquina inferior derecha de la ventana del appletviewer . Coloque el apuntador del ratón sobre la boca de la imagen para escuchar al applet decir “hi” (hola).
JumpingBox	Mueve un rectángulo de manera aleatoria alrededor de la pantalla. ¡Intente atraparlo haciendo clic sobre él con el ratón!
MoleculeViewer	Presenta una vista tridimensional de varias moléculas químicas diferentes. Arrastre el ratón para ver la molécula desde diferentes ángulos.
NervousText	Dibuja texto que salta alrededor de la pantalla.
SimpleGraph	Dibuja una curva compleja.
SortDemo	Compara tres métodos de ordenamiento. Clasifica la información en orden; como palabras en orden alfabético. Cuando usted ejecuta el applet, aparecen tres ventanas del appletviewer . Haga clic en cada una para comenzar el ordenamiento. Observe que los ordenamientos operan a velocidades diferentes.
SpreadSheet	Muestra una hoja de cálculo sencilla con líneas y columnas.
SymbolTest	Dibuja un carácter desde el conjunto de caracteres de Java.
TicTacToe	Permite al usuario jugar Gato en contra de la computadora.
WireFrame	Dibuja una figura en tres dimensiones como una malla alámbrica. Arrastre el ratón para ver la figura desde diferentes ángulos.

Figura 24.9 Ejemplos del directorio **applets**.

Cambie los directorios al subdirectorio **TicTacToe**. En este directorio existe un archivo HTML (**example1.html**) que se utiliza para ejecutar el applet. En la ventana de comando, escriba

```
appletviewer example1.html
```

y oprima la tecla *Entrar*. Esto ejecuta el **appletviewer**. El **appletviewer** carga el archivo HTML especificado como un *argumento de la línea de comando* (**example1.html**), determina desde el archivo cuál applet

cargar (explicaremos los detalles de los archivos HTML en la sección 24.7) y comienza la ejecución del applet. La figura 24.9 muestra varias capturas de pantalla del juego del Gato con este applet.

Tip para prevenir errores 24.3



*Si el comando **appletviewer** no funciona y/o el sistema indica que el comando **appletviewer** no se encuentra, la variable de ambiente **PATH** podría no estar definida apropiadamente en su computadora. Revise las direcciones de instalación para el Java 2 Software Development Kit para asegurarse de que la variable de ambiente está correctamente definida para su sistema (en algunas computadoras, podría ser necesario reiniciar el equipo después de definir la variable de ambiente **PATH**).*

Usted es el jugador **X**. Para interactuar con el applet, apunte el ratón sobre el cuadro en donde desea colocar la **X** y haga clic con el botón del ratón (por lo general, con el botón izquierdo). El applet reproduce un sonido (suponemos que su computadora soporta la reproducción de audio) y coloca una **X** en el cuadrado si éste está vacío. Si el cuadrado está ocupado, éste es un movimiento inválido y el applet ejecuta un sonido diferente que indica que usted no puede realizar ese movimiento específico. Después de hacer un movimiento válido, el applet responde haciendo su propio movimiento (esto sucede de inmediato).

Para jugar de nuevo, ejecute de nuevo el applet haciendo clic en el menú **Subprograma** del **appletviewer**, y seleccionar el elemento de menú **Volver a cargar**. Para finalizar el **appletviewer**, haga clic en el menú **Subprograma** y seleccione el *elemento de menú Salir*.

24.6.2 El applet DrawTest

El siguiente applet que explicaremos le permitirá dibujar líneas y puntos de diferentes colores. Para dibujar, simplemente arrastre el ratón sobre el applet y mantenga oprimido el botón mientras arrastra el ratón. Para este ejemplo, cambie al directorio **applets**, y después al subdirectorio **DrawTest**. En dicho directorio se encuentra el archivo **example1.html** que se utiliza para ejecutar el **applet**. En la ventana de comandos, escriba el comando

```
appletviewer example1.html
```

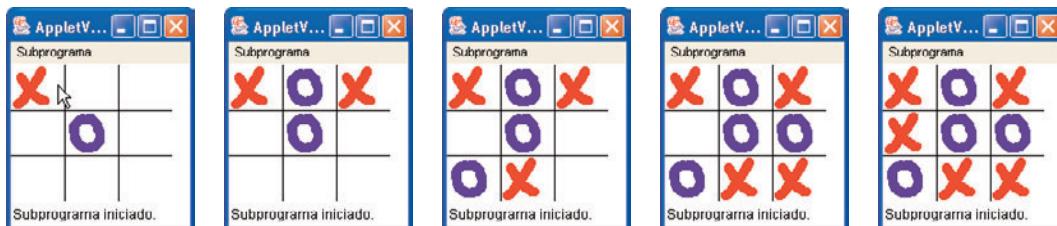
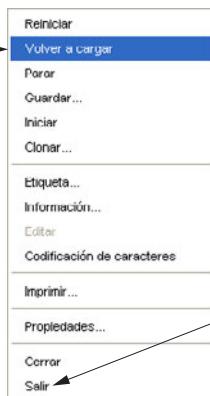


Figura 24.10 Ejecución de ejemplo del applet **TicTacToe**.

Vuelva a **cargar** el applet para ejecutarlo nuevamente



Seleccione **Salir** para finalizar el **appletviewer**

Figura 24.11 Selección de **Volver a cargar** del menú **Subprograma** del **appletviewer**.

y oprima la tecla *Entrar*. Esto ejecuta el **appletviewer**. El **appletviewer** carga el archivo HTML especificado como su argumento en la línea de comando (de nuevo, **example1.html**), determina en el archivo cuál applet cargar y comienza la ejecución del applet. En la figura 24.12 puede apreciar una captura de pantalla de este applet después de dibujar algunas líneas y puntos.

La figura predeterminada a dibujar es una línea y el color predeterminado es el negro, de modo que usted puede dibujar líneas negras al instante con solo arrastrar el ratón a lo largo del applet. Para arrastrar el ratón, presione el botón del ratón, manténgalo presionado y muévalo. Observe que la línea sigue al apuntador del ratón alrededor del applet. La línea no se vuelve permanente hasta que suelta el botón del ratón. Puede comenzar una nueva línea, repitiendo el proceso.

Seleccione un color haciendo clic en el círculo interno de uno de los rectángulos coloreados que se encuentran en la parte inferior del applet. Puede seleccionar entre el rojo, el verde, el azul, el anaranjado y el negro. Por lo general, a los componentes GUI utilizados para presentar estas opciones se les conoce como *botones de opción*. Si se imagina el estéreo de un automóvil, solamente se puede seleccionar una estación de radio a la vez. De manera similar, solamente se puede dibujar en un color a la vez.

Intente modificar la figura de **Líneas a Puntos**, haciendo clic en la flecha que apunta hacia abajo que aparece a la derecha de la palabra **Lines** en la parte inferior del applet. La lista desplegable del componente GUI contiene dos opciones, **Lines** y **Points**. Para seleccionar **Points**, haga clic en la palabra **Points** de la lista. El componente GUI cierra la lista, y ahora **Points** será la figura actual. Por lo general, a este componente GUI se le conoce como *opción, cuadro combinado o lista desplegable*.

Para comenzar un nuevo dibujo, seleccione **Volver a cargar** desde el menú **Subprograma del Appletviewer**. Para finalizar el applet, seleccione **Salir** del menú **Subprograma del appletviewer**.

24.6.3 El applet Java2D

El último applet que explicamos (figura 24.13), antes de definir nuestros propios applets, muestra muchas de las nuevas y complejas capacidades de dos dimensiones incluidas en Java 2; conocida como el *API Java2D*. Para este ejemplo, cambie al directorio **jfc** que se encuentra en el directorio **demo** del J2SDK, después cambie al directorio **Java2D** (usted puede moverse en el árbol de directorios hacia **demo** con el comando “**cd ..**”, tanto

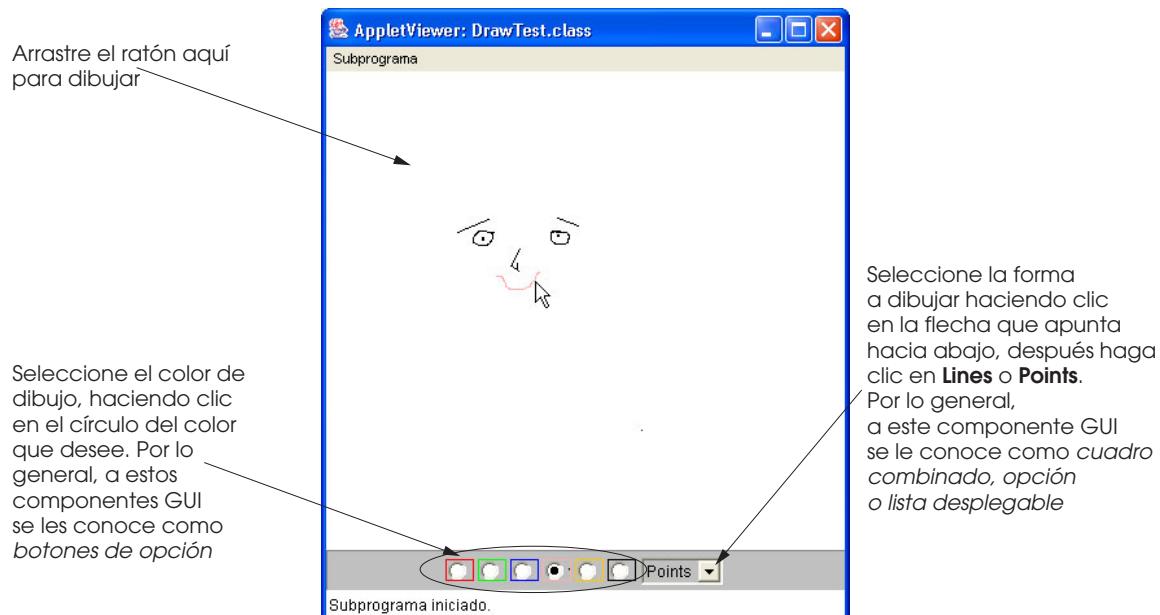


Figura 24.12 Ejemplo de la ejecución del applet **DrawTest**.

en Windows como en UNIX). En dicho directorio se encuentra un archivo HTML (**Java2DemoApplet.html**), que se utiliza para ejecutar el applet. En la ventana de comando escriba

```
appletviewer Java2DemoApplet.html
```

y oprima la tecla *Entrar*. Esto ejecuta el **appletviewer**. El **appletviewer** carga el archivo HTML especificado como su argumento de línea de comando (**Java2DemoApplet.html**), determina desde el archivo cuál applet cargar y comienza la ejecución del applet. Este **demo** en particular se lleva cierto tiempo en cargar, debido a que es bastante grande. La figura 24.12 muestra una captura de pantalla de una de las muchas demostraciones de este applet con respecto a las nuevas capacidades para gráficos de dos dimensiones de Java.

En la parte superior de este demo se aprecian fichas que parecen carpetas de un archivero. Este demo proporciona 11 fichas diferentes con muchas características diferentes en cada ficha. Para cambiar a una parte diferente del demo, simplemente haga clic en una de las fichas. También intente modificar las opciones en la esquina superior derecha del applet. Algunas de éstas afectan la velocidad a la cual el applet dibuja los gráficos. Por ejemplo, haga clic en el cuadro pequeño con una marca en él (un componente GUI conocido como *cuadro de verificación*) a la izquierda de la palabra **Anti-Aliasing** para deshabilitar la técnica de distorsión de gráficos (una técnica gráfica para producir gráficos en pantalla más suaves, en los que los límites de las figuras se hacen más difusos). Cuando se deshabilita esta característica (es decir, el *cuadro de verificación* se desmarca), se incrementa la velocidad de animación de las figuras animadas en el fondo del demo que aparece en la figura 24.13. Esto se debe a que una figura animada con distorsión toma más tiempo para dibujarse que una figura animada sin distorsión.

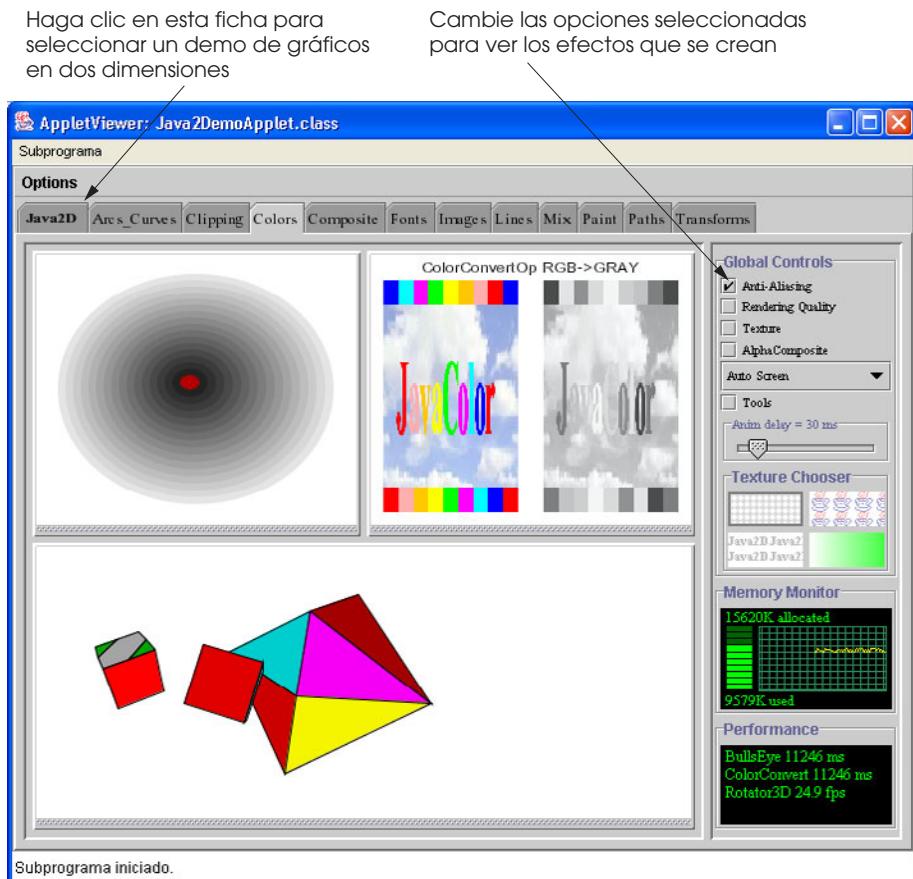


Figura 24.13 Ejecución de ejemplo del applet **Java2D**.

24.7 Un applet sencillo en Java: Cómo dibujar una cadena

Ahora, comenzamos con algunos applets propios. Recuerde que sólo estamos comenzando; tenemos que aprender muchas cosas más antes de que podamos escribir applets similares a las que mostramos en las secciones 24.6. Sin embargo, en capítulos posteriores abordaremos muchas de las mismas técnicas.

Comencemos considerando un applet sencillo que imita el programa de la figura 24.2 al desplegar la cadena **"Bienvenido a la programación en Java!"**. El applet y su salida en la pantalla aparecen en la figura 24.14. La figura 24.15 muestra y explica el documento HTML para cargar el applet en el **applet-viewer**.

```

1 // Figura 24.14: AppletBienvenido.java
2 // Un primer applet en Java
3 import javax.swing.JApplet; // importa la clase JApplet
4 import java.awt.Graphics; // importa la clase Graphics
5
6 public class AppletBienvenido extends JApplet {
7     public void paint( Graphics g )
8     {
9         g.drawString( "Bienvenido a la programacion en Java!", 25, 25 );
10    } // fin del método paint
11 } // fin de la clase AppletBienvenido

```

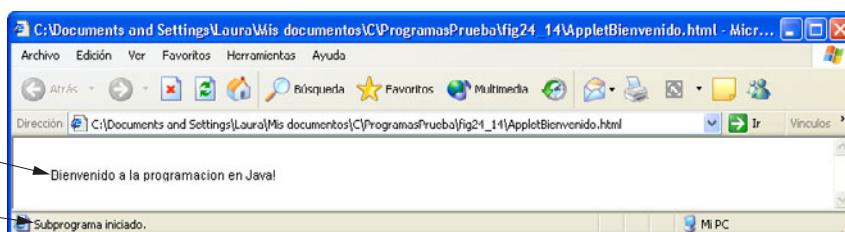
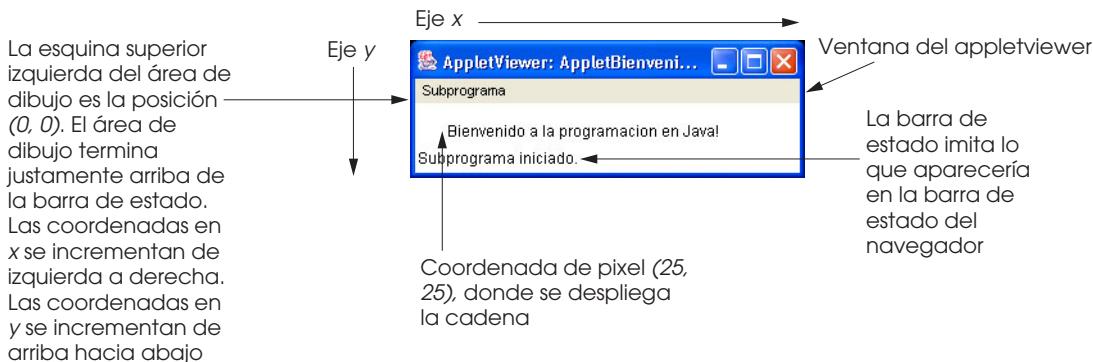


Figura 24.14 Un primer applet en Java, y su salida en pantalla.

```

1 <html>
2 <applet code="AppletBienvenido.class" width=300 height=30>
3 </applet>
4 </html>

```

Figura 24.15 Archivo **AppletBienvenido.html**, el cual carga la clase **AppletBienvenido** de la figura 24.14 dentro del **appletviewer**.

Este programa muestra muchas características importantes de Java. Consideraremos con detalle cada línea del programa. La línea 9 hace el “trabajo real”, a saber: el dibujo de la cadena **Bienvenido a la programación en Java!** en la pantalla. Sin embargo, consideremos cada línea del programa en orden. Las líneas 1 y 2

```
// Figura 24.14: AppletBienvenido.java
// Un primer applet en Java
```

comienzan con //, lo que indica que el resto de cada línea es un comentario. El comentario en la línea 1 indica el número de la figura y el nombre del código fuente del applet. El comentario **Un primer applet en Java** de la línea 2 simplemente describe el propósito del programa.

Como dijimos anteriormente, Java contiene muchas piezas predefinidas llamadas clases (o tipos de datos) que están agrupadas dentro de paquetes en el API de Java. Las líneas 3 y 4

```
import javax.swing.JApplet;           // importa la clase JApplet
import java.awt.Graphics;             // importa la clase Graphics
```

son instrucciones para importar que le indican al compilador en dónde localizar las clases requeridas para compilar este applet de Java. Estas líneas específicas le indican al compilador que la clase **JApplet** se encuentra ubicada en el paquete **javax.swing** y que la clase **Graphics** se encuentra ubicada en **java.awt**. Cuando usted crea un applet de Java, por lo general importa la clase **JApplet**. Importamos la clase **Graphics** para que el programa pueda dibujar gráficos (tales como líneas, rectángulos, elipses y cadenas de caracteres) en un applet de Java (o en una aplicación, más adelante en el libro). [Nota: Existe una clase más antigua llamada **Applet** del paquete **java.applet** que no se utiliza con los componentes GUI más novedosos del paquete **javax.swing**. En el libro, solamente utilizaremos la clase **JApplet** para los applets.]

Cada pieza del nombre del paquete es un directorio (o carpeta) en el disco. Todos los paquetes en el API de Java se almacenan en el directorio **java** o **javax** que contiene muchos subdirectorios, incluso **awt** y **swing**. [Nota: Si busca estos directorios en el disco, no los encontrará debido a que están almacenados dentro de un archivo comprimido especial llamado *Java archive file (JAR)*. Dentro de la estructura de directorios de instalación de J2SDK es un archivo llamado **rt.jar** que contiene los archivos **.class** para el API completo de Java.]

Así como las aplicaciones, cada applet de Java está compuesto por al menos una definición de clase. Una característica clave de las definiciones de clases que no mencionamos antes es que usted rara vez creará una definición de clase “desde cero”. De hecho, cuando crea una definición de una clase, por lo general utiliza piezas de una definición de clase existente. Java utiliza la *herencia* (que explicaremos con detalle en el capítulo 20) para crear nuevas clases a partir de definiciones de clases existentes. La línea 6

```
public class AppletBienvenido extends JApplet {
```

inicia la definición de la clase **AppletBienvenido**. Una vez más, la palabra reservada **class** introduce la definición de una clase e inmediatamente es seguido por el nombre de la clase (**AppletBienvenido**, en este caso). La palabra reservada **extends** seguida por el nombre de la clase indica la clase (en este caso **JApplet**), a partir de la cual nuestra nueva clase hereda las piezas existentes. En esta relación de herencia, a **JApplet** se le conoce como la *superclase* o la *clase base*, y a **AppletBienvenido** se le llama la *subclase* o la *clase derivada*. En el capítulo 27 explicaremos con detalle la herencia. Utilizar la herencia en este punto da como resultado una nueva definición de clase que tiene los *atributos* (datos) y *comportamientos* (métodos) de la clase **JApplet**, así como las nuevas características que agregamos en nuestra definición de la clase **AppletBienvenido** (específicamente, la habilidad de desplegar en la pantalla **Bienvenido a la programación en Java!**).

Un beneficio importante de extender la clase **JApplet** es que cualquiera tiene definido “lo que significa un applet”. El **appletviewer** y los navegadores de la World Wide Web que soportan applets esperan que todos los applets de Java tengan ciertas capacidades (atributos y comportamientos), y la clase **JApplet** proporciona todas esas capacidades; los programadores no necesitan definir todas las capacidades por su cuenta (de nuevo, los programadores no necesitan “reinventar la rueda”). De hecho, un applet requiere alrededor de 200 métodos diferentes para completar su definición. Hasta este punto, en nuestros programas hemos definido un método para cada programa. Si hubiéramos tenido que definir cerca de 200 métodos sólo para desplegar **Bienvenidos a la programación en Java!**, ¡probablemente nunca hubiéramos definido un applet! Con tan sólo utilizar **extends** para heredar de la clase **JApplet**, ahora todos los métodos de **JApplet** son parte de nuestra clase **AppletBienvenido**.

El mecanismo de herencia es fácil de utilizar: el programador no necesita conocer cada detalle de la clase **JApplet** o cualquier otra clase a partir de la que hereda. El programador sólo necesita conocer que la clase **JApplet** ya tiene definidas las capacidades requeridas para crear el applet mínimo. Sin embargo, para hacer mejor uso de cualquier clase, el programador debe estudiar todas las capacidades de la clase que se extiende.



Buena práctica de programación 24.15

Investigue cuidadosamente las capacidades de cualquier clase en la documentación del API de Java, antes de heredar a una subclase. Esto ayuda a asegurar que el programador no redefine por descuido una capacidad que ya está proporcionada.

Las clases se utilizan como “plantillas” o “anteproyectos” para *instanciar* (o *crear*) *objetos* que se utilizarán en el programa. Un objeto (o *instancia*) reside en la memoria de la computadora y contiene información que utiliza el programa. Por lo general, el término *objeto* implica que los atributos (datos) y los comportamientos (métodos) están asociados con el objeto. Los métodos del objeto utilizan atributos para proporcionar servicios útiles para el *cliente del objeto* (es decir, el código que llama a los métodos).

Nuestra clase **AppletBienvenido** se utiliza para crear un objeto que implementa los atributos y comportamientos del applet. El comportamiento predeterminado del método **paint** de la clase **JApplet** no hace cosa alguna. La clase **AppletBienvenido** *redefine* (o *reemplaza*) el comportamiento **paint** que dibuja un mensaje en la pantalla. Cuando un **appletviewer** o un navegador le indica al applet que se “dibuje a sí mismo” en la pantalla mediante la llamada al método **paint**, nuestro mensaje **Bienvenido a la programación en Java!** aparece, en vez de una pantalla en blanco.

El **appletviewer** o navegador en el que se ejecuta el applet es responsable de la creación del objeto (instancia) de la clase **AppletBienvenido**. [Nota: Con frecuencia los términos *instancia* y *objeto* se utilizan indistintamente.] La palabra reservada **public** de la línea 6 es necesaria para permitir al navegador la creación de un objeto de la clase **AppletBienvenido** y la ejecución del applet. La clase que hereda de **JApplet** para crear un applet debe ser una clase pública. En el capítulo 26, explicamos con detalle la palabra reservada **public** y las palabras reservadas relacionadas (tales como **private** y **protected**). Por ahora, simplemente le pediremos que comience todas las definiciones de las clases con la palabra reservada **public**, hasta que la expliquemos en el capítulo 26.

Cuando guarda la clase **public** en un archivo, el nombre de la clase se utiliza como parte del nombre del archivo. Para nuestro applet, el nombre del archivo debe ser **AppletBienvenido.java**. Observe que el nombre del archivo debe deletrearse exactamente como en el nombre de la clase y debe tener la extensión de nombre de archivo **.java**.



Tip para prevenir errores 24.4

*El mensaje de error del compilador “Public class ClassName must be defined in a file called ClassName.java” indica que 1) el nombre del archivo no coincide exactamente con el nombre de la clase **public** en el archivo (incluidas todas las letras mayúsculas y minúsculas), o 2) que usted escribió el nombre de la clase de manera incorrecta cuando compiló la clase (el nombre debe deletrearse con las letras mayúsculas y minúsculas apropiadas).*

Al final de la línea 6, la llave izquierda, **{**, comienza el cuerpo de la definición de la clase. La llave derecha correspondiente, **}**, de la línea 11 termina la definición de la clase. La línea 7

```
public void paint( Graphics g )
```

comienza la definición del *método paint* del applet. El método **paint** es uno de los tres métodos (comportamientos) que con seguridad serán llamados cuando comience la ejecución del applet. Estos tres métodos son **init** (que explicaremos más adelante en este capítulo), **start** (que explicaremos más adelante en el libro) y **paint**, y seguramente serán llamados en ese orden. Estos métodos se llaman desde el **appletviewer** o desde el navegador en el que se ejecuta el applet. Su clase applet obtiene una versión “libre” de cada uno de estos métodos desde la clase **JApplet**, cuando usted especifica **extends JApplet** en la primera línea de su definición de la clase applet. Existen muchos otros métodos que seguramente se llamarán durante la ejecución del applet, explicaremos dichos métodos en el capítulo 25.

La versión libre de cada uno de estos tres métodos se define con un cuerpo vacío (es decir, de manera predeterminada, estos métodos no realizan tarea alguna). Una de las razones por las que heredamos todos los applets de la clase **JApplet** es para obtener nuestras copias libres de los métodos que se llaman de manera automática durante la ejecución de un applet (y también muchos otros métodos).

¿Por qué desearía una copia gratis de un método que no hace cosa alguna? La secuencia de inicio predefinida para las llamadas a los métodos hechas por el **appletviewer** o por el navegador para cada applet siempre es **init**, **start** y **paint**; esto proporciona a un programador de applets una secuencia de inicio garantizada para las llamadas a los métodos al comenzar la ejecución de cada applet. No todos los applets necesitan estos tres métodos. Sin embargo, el **appletviewer** o el navegador esperan que cada uno de estos métodos esté definido, de modo que pueda proporcionar una secuencia de inicio consistente para un applet. [Nota: Esto es similar para las aplicaciones que siempre inician la ejecución en **main**.] Heredar las versiones predeterminadas para estos métodos garantiza que el navegador pueda tratar a cada objeto del applet de manera uniforme al llamar a **init**, **start**, y **paint** cuando comience la ejecución de los applets. Además, el programador puede concentrarse sólo en la definición de los métodos requeridos para un applet en especial.

Las líneas 7 a 10 son definiciones de **paint**. La tarea o método **paint** sirve para dibujar gráficos (tal como líneas, elipses y cadenas de caracteres) en la pantalla. La palabra reservada **void** indica que este método no devuelve resultado alguno cuando termina su tarea. El conjunto de paréntesis después de **paint** define la lista de parámetros del método. Recuerde que la lista de parámetros es en donde los métodos reciben los datos necesarios para llevar a cabo su tarea. Por lo general, los datos pasan del programador al método a través de la *llamada al método* (también conocida como *invocación de un método* o *envío de un mensaje*). Por ejemplo, en la figura 24.4, pasamos los datos a **JOptionPane.showMessageDialog**, incluso el mensaje a desplegar y el tipo de diálogo de mensaje. Sin embargo, el método **paint**, el cual es llamado para que podamos dibujar en el área de visualización del applet en la pantalla, recibe automáticamente la información necesaria cuando se llama al método. La lista de parámetros del método **paint** indica que requiere un objeto **Graphics** (llamado **g**) para realizar su tarea. El objeto **Graphics** se utiliza en **paint** para dibujar gráficos sobre el applet. La palabra reservada **public** al principio de la línea 7 es necesaria para que el **appletviewer** o el navegador puedan llamar a su método **paint**. Por ahora, todas las definiciones de métodos deben comenzar con la palabra reservada **public**. En el capítulo 26 presentaremos otras alternativas.

La llave izquierda, {, de la línea 8 comienza la definición del cuerpo del método. La llave derecha correspondiente, }, de la línea 10 termina la definición del cuerpo del método. La línea 9

```
g.drawString( "Bienvenido a la programacion en Java!", 25, 25 );
```

es una instrucción que indica a la computadora que realice una acción (o tarea), a saber, para desplegar los caracteres de la cadena de caracteres **Bienvenido a la programacion en Java!** en el applet. Esta instrucción utiliza el método **drawString** definido por la clase **Graphics** (esta clase define todas las capacidades para dibujar gráficos de un programa en Java, como el dibujo de cadenas de caracteres y el dibujo de figuras tales como rectángulos, elipses y líneas). El método **drawString** se llama mediante el uso del objeto **g** de **Graphics** (en la lista de parámetros de **paint**) seguido por el operador punto (.), seguido por el nombre del método **drawString**. El nombre del método es seguido por un conjunto de paréntesis que contienen la lista de argumentos que **drawString** necesita para realizar su tarea.

El primer argumento para **drawString** es el **String** a dibujar. Los dos últimos argumentos en la lista, **25** y **25**, son las *coordenadas* (o la *posición*) en la que debe dibujarse la esquina inferior izquierda de la cadena en el área de pantalla del applet. Las coordenadas se miden en *píxeles* a partir de la esquina superior izquierda del applet (la esquina superior izquierda del área blanca correspondiente a la pantalla de captura de la figura 24.14).

Un píxel (“elemento de dibujo”) es la unidad de despliegue para la pantalla de su computadora. En una pantalla a color, un píxel aparece como un punto de color en la pantalla. Muchas computadoras personales tienen 640 pixeles para el ancho de la pantalla y 480 pixeles de alto, lo que da un total de 640 por 480 o 307,200 pixeles desplegables. Muchas pantallas de computadora tienen mejores resoluciones de pantalla, es decir, tiene más pixeles para el ancho y la altura de la pantalla. Mientras más alta sea la resolución de la pantalla, más pequeño parece el applet en la pantalla. Los métodos de dibujo de la clase **Graphics** requieren coordenadas para especificar en dónde dibujar sobre el applet (más adelante en el libro mostraremos el dibujo en las aplicaciones). La primera coordenada es la *coordenada x* (el número de pixeles desde el lado izquierdo del applet), y la segunda coordenada es la *coordenada y* (que representa el número de pixeles desde la parte superior del applet).

Cuando se ejecuta la instrucción anterior, ésta dibuja el mensaje **Bienvenido a la programacion en Java!** en el applet en las coordenadas **25** y **25**. Observe que las comillas que encierran a la cadena de caracteres *no* se despliegan en la pantalla.

Después de definir la clase **AppletBienvenido** y de guardar el archivo **AppletBienvenido.java**, la clase debe compilarse con el compilador de Java **javac**. En la ventana de comandos, escriba el comando

```
javac AppletBienvenido.java
```

para compilar la clase **AppletBienvenido**. Si no existen errores de sintaxis, los bytecodes resultantes se almacenan en el archivo **AppletBienvenido.class**.

Después de compilar el programa de la figura 24.14, debemos crear un archivo *HTML* (Lenguaje de Marcación de Hipertexto) para cargar el applet dentro del **appletviewer** (o del navegador) para ejecutarlo. Por lo general, un archivo HTML termina con la extensión de archivo **.html** o **.htm**. Los navegadores despliegan el contenido de los documentos que contienen texto (también conocidos como *archivos de texto*). Para ejecutar un applet de Java, debe proporcionar un archivo de texto HTML que indique cuál applet debe cargar el **appletviewer** (o el navegador) para su ejecución. La figura 24.15 contiene un archivo HTML sencillo, **AppletBienvenido.html**, que se utiliza para cargar el applet dentro del **appletviewer** (o del navegador) definido en la figura 24.14. [Nota: En este libro, siempre mostramos los applets con el **appletviewer**.]

Buena práctica de programación 24.16



Siempre pruebe el applet en el **appletviewer** y asegúrese de que se ejecuta correctamente, antes de cargar el applet en un navegador de la World Wide Web. Con frecuencia, los navegadores guardan una copia de un applet en la memoria hasta que termina la sesión actual de navegación (es decir, hasta que se cierran todas las ventanas del navegador). Por lo tanto, si usted modifica un applet, recomílelo, y luego recargue el applet en el navegador; es probable que no vea los cambios, debido a que el navegador aún está ejecutando la versión original del applet. Cierre todas las ventanas de su navegador para eliminar de la memoria la versión anterior del applet. Abra una nueva ventana del navegador y cargue el applet para ver los cambios.

Observación de ingeniería de software 24.2



Si su navegador Web no soporta Java 2, la mayoría de los applets de este libro no se ejecutarán en su navegador. Esto se debe a que la mayoría de los applets de este libro utilizan las características que son nuevas en Java 2, o a que no se proporcionan con los navegadores que soportan Java 1.1.

Muchos códigos en HTML (o *etiquetas*) vienen en pares. Por ejemplo, las líneas 1 y 4 de la figura 24.15 indican el principio y el final, respectivamente, de las etiquetas HTML en el archivo. Todas las etiquetas HTML comienzan con la *llave angular izquierda*, **<** y terminan con una *llave angular derecha*, **>**. Las líneas 2 y 3 son etiquetas especiales en HTML para los applets de Java. Éstas le indican al **appletviewer** (o al navegador) que cargue un applet específico y que defina el tamaño del área de despliegue del applet (su *ancho* y su *altura* en pixeles) en el **appletviewer** (o el navegador). Por lo general, el applet y su archivo HTML correspondiente se almacenan en el mismo directorio en el disco.

Por lo general, un archivo HTML se carga en el navegador desde una computadora conectada a Internet diferente a la suya. Sin embargo, los archivos HTML también pueden residir dentro de su computadora (como lo demostraremos en la sección 24.6). Siempre que se carga un archivo HTML que especifica la ejecución de un applet dentro del **appletviewer** (o del navegador), el **appletviewer** (o el navegador) carga el archi-

vo (o archivos) **.class** del applet desde el mismo directorio en la computadora desde la que se cargó el archivo HTML.

La etiqueta **<applet>** tiene diversos componentes. El primer componente de la etiqueta **<applet>** de la línea 2 (**código="AppletBienvenido.class"**) indica que el archivo **AppletBienvenido.class** contiene la clase compilada del applet. Recuerde, cuando compila sus programas en Java, cada clase se compila en un archivo aparte que tiene el mismo nombre que la clase, y termina con la extensión **.class**. El segundo y el tercer componente de la etiqueta **<applet>** indican el **ancho (width)** y la **altura (height)** del applet en pixeles. La esquina superior izquierda del área de visualización siempre es la coordenada 0 en *x*, y la coordenada 0 en *y*. El ancho de este applet es de 300 pixeles. Usted podría querer (o necesitar) utilizar valores más grandes para el ancho y la altura para definir un área de dibujo más grande para sus applets. En la línea 3, la etiqueta **</applet>** finaliza la etiqueta **<applet>** que comenzó en la línea 2. En la línea 4, la etiqueta **</html>** especifica el final de las etiquetas HTML que comienzan en la línea 1 con **<html>**.

Observación de ingeniería de software 24.3

Por lo general, cada applet debe tener un tamaño menor a 640 pixeles de ancho y 480 pixeles de alto (la mayoría de las pantallas de computadora soportan estas dimensiones de ancho y altura mínimas).

Error común de programación 24.11

Colocar caracteres adicionales tales como comas (,) entre los componentes de la etiqueta <applet> puede provocar que el appletviewer o el navegador produzcan un mensaje de error que indica un MissingResourceException al cargar el applet.

Error común de programación 24.12

Olvidar la etiqueta </applet> provoca la carga incorrecta del applet dentro del appletviewer o el navegador.

Tip para prevenir errores 24.5

Si usted recibe un mensaje de error MissingResourceException durante la carga de un applet dentro del appletviewer o del navegador, verifique cuidadosamente la etiqueta <applet> en el archivo HTML para ver si hay de errores de sintaxis. Compare su archivo HTML con el archivo de la figura 24.15 para confirmar una sintaxis adecuada.

El **appletviewer** solamente comprende las etiquetas **<applet>** y **</applet>** de HTML, de modo que en ocasiones se le conoce como el “navegador mínimo” (ignora todas las demás etiquetas de HTML). El **appletviewer** es el lugar ideal para probar la ejecución de un applet y garantizar que dicho applet se ejecuta apropiadamente. Una vez que verifica la ejecución del applet, usted puede agregar las etiquetas **<applet>** y **</applet>** al archivo HTML que será visto por la gente que navega en Internet. Para ejecutar el **AppletBienvenido**, el **appletviewer** se invoca desde la ventana de comandos de la siguiente manera:

```
appletviewer AppletBienvenido.html
```

Observe que el **appletviewer** requiere un archivo HTML para cargar un applet. Esto difiere del intérprete **java** para aplicaciones que requieren que el nombre de la clase sea el mismo que el de la aplicación. Además, debe emitirse el comando anterior desde el directorio en el que se localiza el archivo HTML y el archivo **.class** del applet.

Error común de programación 24.13

Ejecutar el appletviewer con un nombre de archivo que no termina con .html o .htm provoca un error que evita que el appletviewer cargue su applet para ejecución.

Tip de portabilidad 24.2

Verifique sus applets en todos los navegadores utilizados por la gente que ve su applet. Esto le ayudará a asegurar que la gente que vea su applet experimente la funcionalidad que usted espera. [Nota: Una meta del plug-in de Java (que explicaremos posteriormente) es proporcionar la ejecución consistente de un applet en diferentes navegadores.]

24.8 Dos ejemplos más de applets: Cómo dibujar cadenas y líneas

Consideremos ahora otro applet. **Bienvenido a la programación en Java!** puede desplegarse de diferentes maneras. Dos instrucciones **drawString** en el método **paint** puede imprimir varias líneas como en la figura 24.16 (el archivo HTML correspondiente se encuentra en la figura 24.17).

Observe que cada **drawString** puede dibujar en cualquier píxel sobre el applet. La razón por la que las líneas de salida aparecen como muestra la ventana de salida es que especificamos la misma coordenada *x* (25) para cada **drawString**, de modo que las cadenas aparecen alineadas al lado izquierdo, y especificamos coordenadas y diferentes (25 en la línea 9 y 40 en la línea 10), de modo que las cadenas aparecen en ubicaciones diferentes en el applet. Si invertimos las líneas 9 y 10 en el programa, la salida también aparecerá como se muestra, ya que las coordenadas de los pixeles especificados en cada instrucción **drawString** son completamente independientes de las coordenadas especificadas en todas las demás instrucciones **drawStrings** (y todas las operaciones de dibujo). El concepto de líneas de texto como mostramos en los métodos **System.out.println** y **JOptionPane.showMessageDialog** no existe al dibujar gráficos. De hecho, si usted intenta desplegar una cadena que contiene un carácter nueva línea (\n), solamente verá una pequeña caja negra en la posición de la cadena.

Para hacer más interesante el dibujo, el applet de la figura 24.18 dibuja dos líneas y una cadena. El archivo HTML para cargar el applet dentro del **appletviewer** aparece en la figura 24.19.

Las líneas 9 y 10 del método **paint**

```
g.drawLine( 15, 10, 210, 10);
g.drawLine( 15, 30, 210, 30);
```

```

1 // Figura 24.16: AppletBienvenido2.java
2 // Cómo desplegar varias cadenas
3 import javax.swing.JApplet; // importa la clase JApplet
4 import java.awt.Graphics; // importa la clase Graphics
5
6 public class AppletBienvenido2 extends JApplet {
7     public void paint( Graphics g )
8     {
9         g.drawString( "Bienvenido a", 25, 25 );
10        g.drawString( "la programacion en Java !", 25, 40 );
11    } // fin del método paint
12 } // fin de la clase AppletBienvenido2

```

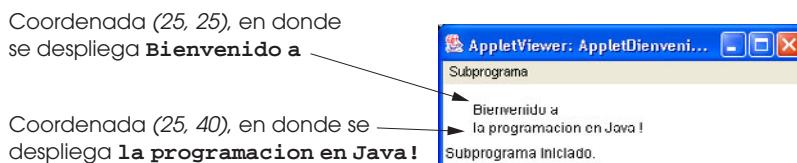


Figura 24.16 Cómo desplegar varias cadenas.

```

1 <html>
2 <applet code="AppletBienvenido2.class" width=300 height=45>
3 </applet>
4 </html>

```

Figura 24.17 Archivo **AppletBienvenido2.html**, el cual carga la clase **AppletBienvenido2** de la figura 24.16 dentro del **appletviewer**.

```

1 // Figura 24.18: LineasBienvenido.java
2 // Cómo desplegar texto y líneas
3 import javax.swing.JApplet; // importa la clase JApplet
4 import java.awt.Graphics; // importa la clase Graphics
5
6 public class LineasBienvenido extends JApplet {
7     public void paint( Graphics g )
8     {
9         g.drawLine( 15, 10, 210, 10 );
10        g.drawLine( 15, 30, 210, 30 );
11        g.drawString( "Bienvenido a la programacion en Java !", 25, 25 );
12    } // fin del método paint
13 } // fin de la clase LineasBienvenido

```

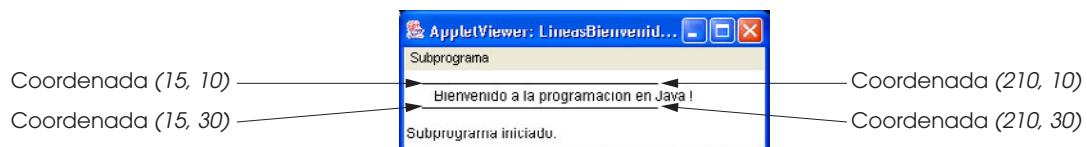


Figura 24.18 Cómo dibujar cadenas y líneas.

```

1 <html>
2 <applet code="LineasBienvenido.class" width=300 height=40>
3 </applet>
4 </html>

```

Figura 24.19 Archivo **LineasBienvenido.html**, el cual carga la clase **LineasBienvenido** de la figura 24.18 en el **appletviewer**.

utilizan el *método drawLine* de la clase **Graphics** para indicar que el objeto **Graphics** al que hace referencia **g** debe dibujar líneas. El método **drawLine** requiere cuatro argumentos para representar los dos puntos finales de la línea sobre el applet, la coordenada *x* y la coordenada *y* del primer punto final en la línea, y la coordenada *x* y la coordenada *y* del segundo punto final en la línea. Todos los valores coordinados se especifican con respecto a la coordenada de la esquina superior izquierda (*0, 0*) del applet. Cuando se llama al método **drawLine**, éste simplemente dibuja una línea entre dos puntos específicos.

24.9 Otro applet de Java: Suma de enteros

Nuestro siguiente applet (figura 24.20) imita la aplicación de la figura 24.7 para sumar dos enteros. Sin embargo, este applet requiere que el usuario introduzca dos *números de punto flotante* (es decir, números con un punto decimal tal como 7.33, 0.0975 y 1000.12345). Para almacenar en memoria números de punto flotante introducimos tipos de datos primitivos **double**, los cuales se utilizan para representar *números de punto flotante* de doble precisión. También existen tipos de datos primitivos **float** para almacenar *números de punto flotante de precisión sencilla*. Un **double** requiere más memoria para almacenar un valor de punto flotante, pero lo almacena con aproximadamente el doble de precisión que un **float** (15 dígitos significativos para **double** contra siete dígitos significativos para **float**).

```

1 // Figura 24.20: AppletSuma.java
2 // Suma de dos números de punto flotante
3 import java.awt.Graphics; // importa la clase Graphics

```

Figura 24.20 Un programa de suma “en acción”.(Parte 1 de 2)

```

4 import javax.swing.*;           // importa el paquete javax.swing
5
6 public class AppletSuma extends JApplet {
7     double suma; // suma de los valores introducidos por el usuario
8
9     public void init()
10    {
11         String primerNumero,      // primera cadena introducida por el usuario
12             segundoNumero;      // segunda cadena introducida por el usuario
13         double numero1,          // primer número a sumar
14             numero2;            // segundo número a sumar
15
16         // lee el primer número del usuario
17         primerNumero =
18             JOptionPane.showInputDialog(
19                 "Introduzca el primer valor de punto flotante" );
20
21         // lee el segundo número del usuario
22         segundoNumero =
23             JOptionPane.showInputDialog(
24                 "Introduzca el segundo valor de punto flotante" );
25
26         // convierte los números del tipo String a tipo double
27         numero1 = Double.parseDouble( primerNumero );
28         numero2 = Double.parseDouble( segundoNumero );
29
30         // suma los números
31         suma = numero1 + numero2;
32     } // fin del método init
33
34     public void paint( Graphics g )
35     {
36         // dibuja los resultados con g.drawString
37         g.drawRect( 15, 10, 270, 20 );
38         g.drawString( "La suma es " + suma, 25, 25 );
39     } // fin del método paint
40 } // fin de la clase AppletSuma

```

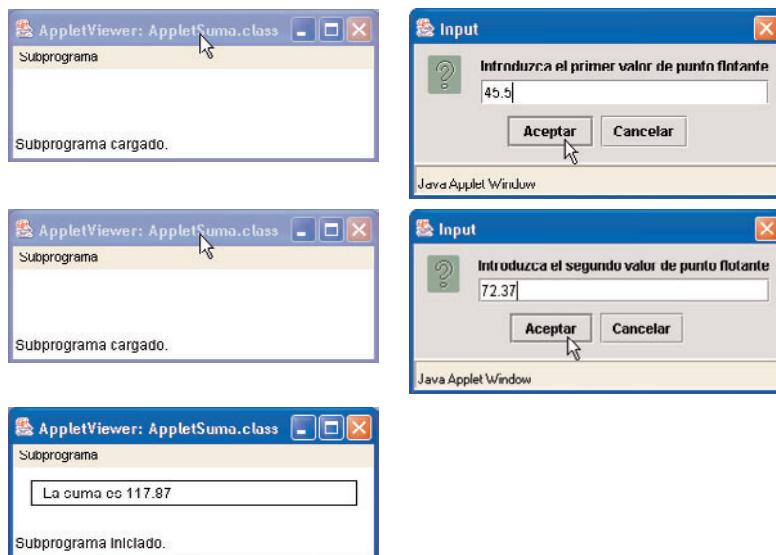


Figura 24.20 Un programa de suma “en acción”. (Parte 2 de 2.)

```

1 <html>
2 <applet code="AppletSuma.class" width=300 height=50>
3 </applet>
4 </html>
```

Figura 24.21 Archivo **AppletSuma.html**, el cual carga la clase **AppletSuma** de la figura 24.20 dentro del **appletviewer**.

Una vez más, utilizamos **JOptionPane.showInputDialog** para solicitar información al usuario. El applet después calcula la suma de los valores de entrada y despliega el resultado dibujando una cadena dentro de un rectángulo en un applet. El archivo HTML para cargar este applet dentro del **appletviewer** aparece en la figura 24.21.

La línea 3

```
import java.awt.Graphics; // importa la clase Graphics
```

especifica al compilador en dónde localizar la clase **Graphics** (del paquete **java.awt**) para utilizarla en esta aplicación. En realidad, la instrucción **import** de la línea 3 no es necesaria, si siempre utilizamos el nombre completo de la clase **Graphics**, **java.awt.Graphics**, el cual incluye el nombre completo del paquete y el nombre de la clase. Por ejemplo, la primera línea del método **paint** puede definirse como:

```
public void paint( java.awt.Graphics g )
```

Observación de ingeniería de software 24.4



*El compilador de Java no necesita instrucciones **import** en un archivo de código fuente de Java si el nombre completo de la clase, es decir, el nombre completo del paquete y el nombre de la clase (por ejemplo, **java.awt.Graphics**), se especifica cada vez que se utiliza el nombre de la clase en el código fuente.*

La línea 4

```
import javax.swing.*; // importa el paquete javax.swing
```

especifica al compilador en dónde se ubica el paquete completo **javax.swing**. El asterisco (*) indica que todas las clases en el paquete **javax.swing** (tal como **JApplet** y **JOptionPane**) deben estar disponibles para el compilador, de modo que éste pueda garantizar que utilizamos las clases de manera correcta. Esto permite a los programadores utilizar el *nombre corto* (el nombre de la clase por sí mismo) de cualquier clase del paquete **javax.swing** dentro del programa. Recuerde que nuestros dos últimos programas solamente soportan la clase **JApplet** del paquete **javax.swing**. Importar un paquete completo dentro de un programa también es una notación abreviada para que el programador no tenga que proporcionar una instrucción **import** para cada clase del paquete. Recuerde que siempre puede utilizar el nombre completo de cada clase, es decir, **javax.swing.JApplet** y **javax.swing.JOptionPane**, en lugar de instrucciones **import**.

Observación de ingeniería de software 24.5



*El compilador no carga cada clase en un paquete cuando encuentra una instrucción **import** que utiliza la notación * (por ejemplo, **javax.swing.***) para indicar que se utilizan diversas clases del paquete dentro del programa. El compilador busca en el paquete solamente las clases que utiliza el programa.*

Observación de ingeniería de software 24.6



*Muchos directorios de paquetes tienen subdirectorios. Por ejemplo, el directorio del paquete **java.awt** contiene el subdirectorio **event** para el paquete **java.awt.event**. Cuando el compilador encuentra una instrucción **import** que utiliza la notación * (por ejemplo, **java.awt.***) para indicar que se utilizan distintas clases del paquete dentro del programa, el compilador no busca el subdirectorio **event**. Esto significa que usted no puede definir un **import** de **java.*** para buscar las clases de todos los paquetes.*

Observación de ingeniería de software 24.7



*Cuando utilice instrucciones **import**, debe especificar instrucciones **import** separadas para cada paquete utilizado en el programa.*

Error común de programación 24.14



*Asumir que una instrucción **import** para un paquete completo (por ejemplo, **java.awt.***) también importa las clases de los subdirectorios de dicho paquete (por ejemplo, **java.awt.event.***), provoca errores de sintaxis para las clases de los subdirectorios. Debe haber un **import** separado para cada paquete del que se utilizan las clases.*

Recuerde que los applets heredan de la clase **JApplet**, de modo que contienen todos los métodos requeridos por el **appletviewer** o el navegador para ejecutar el applet. La línea 6

```
public class AppletSuma extends JApplet {
```

inicia la definición de la clase **AppletSuma** e indica que hereda de **JApplet**.

Todas las definiciones de las clases inician con una llave izquierda de apertura (fin de la línea 6), {, y terminan con una llave derecha de cierre, }, (Línea 40).

Error común de programación 24.15



Si las llaves no se encuentran como pares coincidentes, el compilador indica un error de sintaxis.

La línea 7

```
double suma; // suma de los valores introducidos por el usuario
```

es una *declaración de variable de instancia*; cada instancia (objeto) de la clase contiene una copia de cada variable de instancia. Por ejemplo, si existen 10 instancias en ejecución de este objeto, cada instancia contiene su propia copia de **suma**. Además, existirán 10 copias distintas de **suma** (una para cada applet). Las variables de instancia se declaran en el cuerpo de la definición de la clase, pero no en el cuerpo de cualquier método de la definición de la clase. La declaración anterior establece que **suma** es una variable del tipo primitivo **double**.

Uno de los beneficios importantes de las variables de instancia es que sus identificadores pueden utilizarse a través de la definición de la clase (es decir, en todos los métodos de la clase). Hasta ahora, declaramos todas las variables en el método **main** de una aplicación. A las variables definidas en el cuerpo de un método se les conoce como *variables locales* y solamente pueden utilizarse en el cuerpo del método en el que se definen. Otra diferencia entre las variables de instancia y las variables locales es que a las variables de instancia siempre se les asigna un valor predeterminado y a las variables locales no. La variable **suma** se inicializa automáticamente en 0.0, debido a que es una variable de instancia.

Error común de programación 24.16



Utilizar una variable local no inicializada, es un error de sintaxis. A todas las variables locales se les debe asignar un valor antes de intentar utilizar el valor de dicha variable.

Buena práctica de programación 24.17



Inicializar las variables de instancia en lugar de confiar en la inicialización automática, mejora la legibilidad del programa.

Este applet contiene dos métodos: **init** (definición en las líneas 9 a 32) y **paint** (definición en las líneas 34 a 39). El método **init** es un método especial del applet que por lo general es el primer método definido por el programador en un applet, y con certeza es el primer método del applet en ejecutarse. El método **init** se llama una vez durante la ejecución del applet. Por lo general el método **inicializa** las variables de instancia del applet (si requieren inicializarse en un valor diferente de su valor predeterminado), y realiza cualquier tarea una vez que inicia la ejecución del applet.

Observación de ingeniería de software 24.8



El orden en que se definen los métodos en la definición de una clase no tiene efecto en cuanto al orden en el que se llaman en tiempo de ejecución.

La primera línea en el método **init** siempre aparece como

```
public void init()
```

la cual indica que **init** es un método público que no devuelve información (**void**) cuando completa su tarea, y no recibe argumentos (paréntesis vacíos después de **init**).

La llave izquierda (línea 10) marca el inicio del cuerpo de **init**, y la llave derecha correspondiente (línea 32) marca el final de **init**. Las líneas 11 y 12

```
String primerNumero,           // primera cadena introducida por el usuario
      segundoNumero;          // segunda cadena introducida por el usuario
```

son la declaración para las variables locales **primerNumero** y **segundoNumero** de tipo **String**.

Las líneas 13 y 14

```
double numero1,               // primer número a sumar
      numero2,               // segundo número a sumar
```

declaran que las variables **numero1** y **numero2** son tipos de datos primitivos **double**, lo cual significa que estas variables almacenan valores de punto flotante. Éstas son variables de instancia, de modo que se inicializan automáticamente en 0.0 (el valor predeterminado para las variables de instancia **double**).

Observe que en realidad existen dos tipos de variables en Java, *variables de tipos de datos primitivos* (por lo general llamadas *variables*) y *variables de referencia* (por lo general llamadas *referencias*). Los identificadores **primerNumero** y **segundoNumero** son en realidad referencias; nombres utilizados para hacer *referencia a objetos* en el programa. Dichas referencias en realidad contiene la ubicación de un objeto dentro de la memoria de la computadora. En los applets anteriores, el método **paint** en realidad recibe una referencia llamada **g** que hace referencia al objeto **Graphics**. La referencia se utiliza para enviar mensajes al (es decir, llamar métodos de) objeto **Graphics** en memoria que nos permite dibujar sobre un applet. Por ejemplo, la instrucción

```
g.drawString( "Bienvenido a la programacion en Java!", 25, 25 );
```

envía el mensaje **drawString** (llama al método **drawString**) al objeto **Graphics** al que hace referencia. Como parte del mensaje (llamada al método), proporcionamos los datos que requiere **drawString** para llevar a cabo su tarea. El objeto **Graphics** después dibuja el **String** en la ubicación especificada.

Los identificadores **numero1**, **numero2** y **suma** son los nombres de las *variables*. Una variable es similar a un objeto. La principal diferencia entre una variable y un objeto es que un objeto se define mediante la definición de una clase que puede contener tanto datos (variables de instancia) como métodos, mientras que una variable se define mediante un *tipo de dato primitivo (o predefinido)* (de tipo **char**, **byte**, **short**, **int**, **long**, **float**, **double** o **boolean**) que sólo puede contener datos. Una variable puede almacenar exactamente un valor a la vez, mientras que un objeto puede contener muchas piezas individuales de datos. La diferencia entre una variable y una referencia se basa en el tipo de dato del identificador (como se establece en la declaración). Si el tipo de dato es un nombre de clase, el identificador es una referencia a un objeto y dicha referencia puede utilizarse para enviar mensajes al objeto (llamar a los métodos). Si el tipo de dato es uno de los tipos de datos primitivos, el identificador es una variable que puede utilizarse para almacenar en memoria o para recuperar desde la memoria un valor individual del tipo de dato primitivo declarado.

Observación de ingeniería de software 24.9



Una pista para ayudarle a determinar si un identificador es una variable o una referencia es el tipo de dato de la variable. Por convención, todas las clases en Java comienzan con una letra mayúscula. Por lo tanto, si el tipo de dato comienza con una letra mayúscula, por lo general usted puede asumir que el identificador es una referencia a un objeto del tipo declarado (por ejemplo, **Graphics g** indica que **g** es una referencia a un objeto **Graphics**).

Las líneas 16 a 19

```
// lee el primer número del usuario
primerNumero =
    JOptionPane.showInputDialog(
        "Introduzca el primer valor de punto flotante" );
```

leen el primer número de punto flotante del usuario. El método **JOptionPane.showInputDialog** despliega un diálogo de entrada que indica al usuario que introduzca un valor. El usuario escribe un valor en el campo de texto del diálogo de entrada, y luego hace clic en el botón **Aceptar** para devolver la cadena que escribió. [Si usted escribe y no aparece cosa alguna en el campo de texto, coloque el apuntador del ratón en el campo de texto y haga clic para activar el campo de texto.]

Técnicamente, el usuario puede digitar cualquier cosa que desee. En este programa, si el usuario escribe un valor no numérico o hace clic en el botón **Cancelar**, ocurrirá un error en tiempo de ejecución y el mensaje se desplegará en la ventana de comando desde la que se ejecuta el **appletviewer**.

A la variable **primerNumero** se le asigna el resultado de la llamada a la operación **JOptionPane.showInputDialog** en la instrucción de asignación. La instrucción se lee como “**primerNumero** obtiene el valor de **JOptionPane.showInputDialog(“Introduzca el primer valor de punto flotante”)**”.

Las líneas 21 a 24

```
// lee el segundo número del usuario
segundoNumero =
    JOptionPane.showInputDialog(
        "Introduzca el segundo valor de punto flotante" );
```

lee el segundo valor de punto flotante del usuario, desplegando un cuadro de entrada.

Las líneas 26 a 28

```
// convierte los números del tipo String a tipo double
numero1 = Double.parseDouble( primerNumero );
numero2 = Double.parseDouble( segundoNumero );
```

convierte las dos cadenas de entrada del usuario a valores **double** que puede utilizarse en un cálculo. El método **Double.parseDouble** (un método **static** de la clase **Double**) convierte su argumento **String** al valor **double** de punto flotante **Double** que es parte del paquete **java.lang**. El valor de punto flotante devuelto por **Double.parseDouble** en la línea 27 se asigna a la variable **numero1**. Cualquier referencia subsiguiente a **numero1** en el método utiliza este mismo valor de punto flotante. El valor de punto flotante devuelto por **Double.parseDouble** en la línea 28 se asigna a la variable **numero2**. Cualquier referencia subsiguiente a **numero2** en el método utiliza este valor de punto flotante.

Observación de ingeniería de software 24.10



Para cada tipo de dato primitivo (tal como un **int** o un **double**) existe una clase correspondiente (tal como **Integer** o **Double**) en el paquete **java.lang**. Estas clases (por lo general conocidas como envolturas de tipo) proporcionan métodos para procesar valores de tipos de datos primitivos (tales como convertir un **String** a un valor de tipo de dato primitivo, o convertir un valor de tipo de dato primitivo a un **String**). Los tipos de datos primitivos no tienen métodos. Por lo tanto, los métodos relacionados con un tipo de dato primitivo se ubican en la clase envolvente de tipo correspondiente (es decir, el método **parseDouble** que convierte un **String** a un valor **double** se localiza en la clase **Double**).

La instrucción de asignación de la línea 31

```
suma = numero1 + numero2;
```

calcula la suma de las variables **numero1** y **numero2** y asigna el resultado a la variable **suma** por medio del operador **=**. La instrucción se lee como “**suma** obtiene el valor de **numero1 + numero2**”. La mayoría de los cálculos se realizan con instrucciones de asignación. Observe que la variable de instancia **suma** se utiliza en la instrucción anterior en el método **init**, aun cuando **suma** no se definió en el método **init**. Definimos **suma** como una variable de instancia, por lo que podemos utilizar **init** y todos los otros métodos de la clase.

El método **init** del applet retorna y el **appletviewer** o el navegador llama al método **start** del applet. No definimos el método **start** en este applet, por lo que aquí utilizamos el método proporcionado por la clase **JApplet**. El método **start** se utiliza primordialmente con un concepto avanzado llamado subprocesamiento múltiple, el cual no explicamos en estos capítulos introductorios.

A continuación, el navegador llama al método **paint** del applet. En este ejemplo, el método **paint** dibuja un rectángulo que contiene una cadena con el resultado de la suma. La línea 37

```
g.drawRect( 15, 10, 270, 20);
```

envía el mensaje **drawRect** al objeto **Graphics** al que **g** hace referencia (llama al método **drawRect** del objeto **Graphics**). El método **drawRect** dibuja un rectángulo, basándose en sus cuatro argumentos. Los primeros dos valores enteros representan la *coordenada superior izquierda x*, y la *coordenada superior izquierda y*, en donde el objeto **Graphics** comienza el dibujo del rectángulo. El tercero y cuarto argumentos son núme-

ros enteros negativos que representan el *ancho* y la *altura* del rectángulo en píxeles, respectivamente. Esta instrucción en particular dibuja un rectángulo que comienza con la coordenada (15, 10) que es de 270 píxeles de ancho y de 20 píxeles de alto.

Error común de programación 24.17



Proporcionar un ancho negativo o una altura negativa como argumentos del método `drawRect` de `Graphics`, es un error lógico. El rectángulo no se desplegará y no indicará error alguno.

Error común de programación 24.18



Proporcionar dos puntos (es decir, pares de coordenadas x y y) como argumentos del método `drawRect` de `Graphics`, es un error lógico. El tercer argumento debe ser el ancho en píxeles y el cuarto argumento debe ser la altura en píxeles del rectángulo a dibujar.

Error común de programación 24.19



Por lo general, proporcionar argumentos al método `drawRect` de `Graphics` que provoquen que el rectángulo se dibuje fuera del área visible del applet (es decir, el ancho y la altura del applet como se especifica el documento HTML que hace referencia al applet), es un error lógico. Aumente el ancho y la altura del applet en el documento HTML, o pase los argumentos al método `drawRect` que provoca que el rectángulo se dibuje dentro del área visible del applet.

La línea 38

```
g.drawString( "La suma es " + suma, 25, 25 );
```

envía el mensaje `drawString` al objeto `Graphics` al cual hace referencia `g` (llama al método `drawString` del objeto `Graphics`). La expresión

```
"La suma es " + suma
```

de la instrucción anterior utiliza el operador de concatenación + para concatenar la cadena "La suma es " y `suma` (convertida a una cadena) para crear la cadena desplegada por `drawString`. Observe nuevamente que la variable de instancia `suma` de la instrucción anterior se utiliza, incluso si no se define en el método `paint`.

El beneficio de definir `suma` como una variable de instancia es que pudimos asignar a `suma` un valor en `init` y utilizar el valor `suma` en el método `paint` más adelante en el programa. Todos los métodos de la clase son capaces de utilizar las variables de instancia en la definición de la clase.

Observación de ingeniería de software 24.11



Las únicas instrucciones que deben colocarse en el método `init` del applet son aquellas que se relacionan directamente con la inicialización única de las variables de instancia del applet. Los resultados del applet deben desplegarse a través de otros métodos de la clase del applet. Los resultados que involucran el dibujo deben desplegarse desde el método `paint` del applet.

Observación de ingeniería de software 24.12



Las únicas instrucciones que deben colocarse en el método `paint` del applet son aquellas que se relacionan de manera directa con el dibujo (es decir, las llamadas a los métodos de la clase `Graphics`) y con la lógica del dibujo. Por lo general, los cuadros de diálogo no deben desplegarse desde el método `paint` del applet.

En este capítulo introdujimos muchas características importantes de Java, que incluyen aplicaciones, applets, el despliegado de datos en la pantalla, la entrada de datos desde el teclado, la realización de cálculos y la toma de decisiones. En el siguiente capítulo explicaremos algunas de las diferencias entre Java y C/C++, tal como arreglos, operadores y definiciones de métodos. En los siguientes capítulos explicaremos la programación basada en objetos y orientada a objetos, así como los gráficos en Java, interfaces gráficas de usuario (GUIs) y características multimedia.

RESUMEN

- Java es uno de los lenguajes de desarrollo más populares en la actualidad.
- Java fue desarrollado en Sun Microsystems. Sun proporciona una implementación de la plataforma de Java llamada Java 2 Software Development Kit (J2SDK), la cual incluye el conjunto mínimo de herramientas necesarias para escribir programas en Java.

- Java es un lenguaje completamente orientado a objetos con un enorme soporte para las técnicas de ingeniería de software.
- Por lo general, los sistemas en Java constan de varias partes: el lenguaje, la Interfaz de Programación de Aplicaciones de Java (API, Java Applications Programming Interface), y distintas bibliotecas de clases.
- Por lo general, los programas en Java pasan a través de cinco etapas para poder ejecutarse: edición, compilación, carga, verificación y ejecución.
- Los nombres de programas en Java terminan con la extensión **.java**.
- El compilador de Java (**javac**) traduce un programa en Java a bytecodes, el código comprensible para el intérprete de Java. Si un programa se compila correctamente, se produce un archivo con extensión **.class**. Este archivo contiene los bytecodes que se interpretan durante la fase de ejecución.
- Un programa Java primero debe colocarse en memoria, antes de que pueda ejecutarse. Esto se hace por medio del cargador de clases, el cual toma el archivo (o archivos) **.class** que contienen los bytecodes y los transfiere a la memoria. El archivo **.class** puede cargarse desde un disco en su sistema o sobre una red.
- Una aplicación es un programa que se ejecuta por medio del intérprete **java**.
- Un comentario que comienza con // se llama comentario de una sola línea. Los programadores insertan comentarios para documentar los programas y mejorar su legibilidad.
- A una cadena de caracteres contenida entre comillas se le llama cadena, cadena de caracteres, mensaje, o literal de cadena.
- La palabra reservada **class** introduce la definición de una clase y de inmediato le sigue el nombre de la clase.
- Las palabras reservadas (o palabras clave) están reservadas para el uso de Java.
- Por convención, todos los nombres de clases en Java comienzan con una letra mayúscula. Si el nombre de una clase tiene más de una palabra, cada palabra debe comenzar con mayúscula.
- Un identificador consiste en una serie de caracteres que consta de letras, dígitos, guiones bajos (_) y signos de moneda (\$) que no comienza con un dígito, no contiene espacios y no es una palabra reservada.
- Java es sensible a mayúsculas y a minúsculas, es decir, las letras mayúsculas y minúsculas son diferentes.
- Una llave izquierda, {, inicia el cuerpo de la definición de una clase. Su correspondiente llave derecha, }, finaliza la definición de la clase.
- Las aplicaciones en Java comienzan su ejecución en el método **main**.
- La primera línea del método **main** debe definirse como:

```
public static void main( String args[] )
```

- Una llave izquierda, {, comienza el cuerpo de la definición de un método. Su correspondiente llave derecha, }, termina el cuerpo de la definición del método.
- A **System.out** se le conoce como el objeto estándar de salida. **System.out** permite a las aplicaciones de Java desplegar cadenas y otro tipo de información en la ventana de comando desde la cual se ejecuta la aplicación Java.
- La secuencia de escape \n significa nueva línea. Otras secuencias de escape incluyen \t (tabulador), \r (retorno de carro), \\ (diagonal invertida) y \" (comillas dobles).
- El método **println** del objeto **System.out** despliega (o imprime) una línea de información en la ventana de comandos. Cuando **println** completa su tarea, el cursor se posiciona automáticamente al principio de la siguiente línea en la ventana de comando.
- Toda instrucción debe terminar con punto y coma (también conocido como terminador de instrucción).
- La diferencia entre **System.out.print** y **System.out.println** es que **System.out.print** no posiciona el cursor al principio de la siguiente línea en la ventana de comando cuando termina de desplegar su argumento. El siguiente carácter que se despliega en la ventana de comando aparecerá inmediatamente después del último carácter desplegado con **System.out.print**.
- Java contiene muchas clases predefinidas que se agrupan en directorios del disco, dentro de categorías de clases relacionadas llamadas paquetes. A los paquetes en su conjunto se les conoce como la biblioteca de clases de Java o la interfaz de programación de aplicaciones de Java (API de Java).
- La clase **JOptionPane** está definida para nosotros en un paquete llamado **javax.swing**. La clase **JOptionPane** contiene métodos que despliegan un cuadro de diálogo que contiene información.
- El compilador utiliza instrucciones **import** para localizar las clases requeridas para compilar un programa en Java.
- El paquete **javax.swing** contiene muchas clases que ayudan a definir interfaces gráficas de usuario (GUI) para una aplicación. Los componentes GUI facilitan la entrada de datos del usuario y la salida de datos del programa.

- El método **showMessageDialog** de la clase **JOptionPane** requiere dos argumentos. Hasta que expliquemos **JOptionPane** con detalle en el capítulo 29, el primer argumento siempre será la palabra reservada **null**. El segundo argumento es la cadena a desplegar.
- Se llama a un método estático al colocar a continuación del nombre de la clase un punto (.) y el nombre del método.
- El método **exit** de la clase **System** finaliza una aplicación. La clase **System** es parte del paquete **java.lang**. El paquete **java.lang** se importa automáticamente en todos los programas Java.
- Las variables de tipo **int** almacenan valores de tipo entero (es decir, números completos tales como 7, -11, 0, 31914).
- A los tipos tales como **int**, **float**, **double** y **char** con frecuencia se les llama tipos de datos primitivos. Los nombres de tipos primitivos son palabras reservadas del lenguaje de programación Java.
- El método **Integer.parseInt** (un método estático de la clase **Integer**) convierte su argumento de tipo cadena a un entero.
- Java tiene una versión del operador + para la concatenación de cadenas que permite concatenar una cadena y un valor de otro tipo de dato (incluso otra cadena).
- Los nombres de variables corresponden a ubicaciones en la memoria de la computadora. Toda variable tiene un nombre, un tipo, un tamaño y un valor.
- Toda variable declarada en un método debe inicializarse antes de que pueda utilizarse en una expresión.
- Un applet es un programa en Java que puede ejecutarse en el **appletviewer** (una utilidad de prueba para applets que se incluye con J2SDK), o en un navegador de la World Wide Web como Netscape Communicator o el Internet Explorer de Microsoft. El **appletviewer** (o el navegador) ejecuta un applet cuando un documento en Lenguaje de Marcación de Hipertexto (HTML) que contiene un applet se abre en el **appletviewer** (o en el navegador).
- En el **appletviewer**, usted puede ejecutar de nuevo un applet, haciendo clic en el menú **Subprograma** y seleccionando la opción **Volver a cargar**. Para finalizar un applet, haga clic en el menú **Subprograma del appletviewer** y seleccione la opción **Salir**.
- La clase **Graphics** se encuentra localizada en el paquete **java.awt**. Importe la clase **Graphics** para que el programa pueda dibujar gráficos.
- La clase **JApplet** se localiza en el paquete **javax.swing**. Cuando crea un applet en Java, por lo general se importa la clase **JApplet**.
- Cada porción del nombre del paquete es un directorio (o carpeta) en el disco. Todos los paquetes en el API de Java se almacenan en el directorio **java** o **javax**, el cual contiene muchos subdirectorios.
- Java utiliza la herencia para crear nuevas clases a partir de definiciones de clases existentes. La palabra reservada **extends**, seguida por el nombre de la clase, indica la clase a partir de la cual hereda una nueva clase.
- En la relación de herencia, a la clase a continuación de **extends** se le llama superclase o clase base, y a la nueva clase se le llama subclase o clase derivada. El uso de la herencia da como resultado una nueva definición de clase que tiene los atributos (datos) y los comportamientos (métodos) de la superclase, así como las nuevas características adicionadas en la definición de la subclase.
- Uno de los beneficios de extender la clase **JApplet** es que alguien más ya definió lo que “significa un applet”. El **appletviewer** y los navegadores de la World Wide Web que soportan applets esperan que cada applet de Java contenga ciertas capacidades (atributos y comportamientos), y la clase **JApplet** ya proporciona todas esas capacidades.
- Las clases se utilizan como “plantillas” o “anteproyectos” para instanciar (o crear) objetos en memoria a utilizarse dentro de un programa. Un objeto (o instancia) es una región en la memoria de la computadora en la cual la información se almacena para utilizarse en un programa. Por lo general, el término objeto implica que los atributos (datos) y los comportamientos (métodos) se asocian con el objeto, y que dichos comportamientos realizan operaciones en los atributos del objeto.
- El método **paint** es uno de los tres métodos (comportamientos) que seguramente se invocarán automáticamente cuando inicie la ejecución de cualquier applet. Estos tres métodos son **init**, **start** y **paint**, y con seguridad se llamarán en ese orden. Estos métodos se llaman desde el **appletviewer** o desde el navegador en el que se ejecuta el applet.
- El método **drawString** de la clase **Graphics** dibuja una cadena en una ubicación específica del applet. El primer argumento para **drawString** es la **String** (cadena) a dibujar. Los dos últimos argumentos en la lista son las coordenadas (o posiciones) en las cuales se debe dibujar una cadena. Las coordenadas se miden en pixeles desde la esquina superior izquierda del applet.
- Usted debe crear un archivo HTML (Lenguaje de Marcación de Hipertexto) para cargar un applet dentro del **appletviewer** (o navegador) para ejecutarlo.

- Muchos códigos HTML (conocidos como etiquetas) vienen en pares. Las etiquetas de HTML comienzan con una llave angular izquierda <, y terminan con una llave angular derecha >.
- Por lo general, el applet y su correspondiente archivo HTML se almacenan en el mismo directorio del disco.
- El primer componente de la etiqueta <**applet**> indica el archivo que contiene la clase con el applet compilado. El segundo y el tercer componente de la etiqueta <**applet**> indica el ancho (**width**) y la altura (**height**) del applet en pixeles. Por lo general, cada applet debe ser menor a 640 pixeles de ancho y 480 pixeles de alto (la mayoría de las pantallas de computadora soportan estas dimensiones como ancho y altura mínimos).
- El **appletviewer** solamente comprende las etiquetas <**applet**> y </**applet**> de HTML, de modo que en ocasiones se le llama “navegador mínimo” (ignora todas las demás etiquetas de HTML).
- El método **drawLine** de la clase **Graphics** dibuja líneas. El método requiere cuatro argumentos que representan los dos puntos extremos de la línea en un applet, la coordenada x y la coordenada y del primer punto extremo de la línea, y la coordenada x y la coordenada y del segundo punto extremo de la línea. Todos los valores de las coordenadas se especifican con respecto a la coordenada superior izquierda (0, 0) del applet.
- El tipo de dato primitivo **double** almacena números de punto flotante de doble precisión. El tipo de dato primitivo **float** almacena números de punto flotante de precisión simple. Un **double** requiere más memoria de almacenamiento que un valor de punto flotante, pero lo almacena con aproximadamente el doble de precisión que un **float** (15 dígitos significativos para **double** contra siete dígitos significativos para **float**).
- Las instrucciones **import** no son necesarias, si usted siempre utiliza el nombre completo de una clase, incluyendo el nombre completo del paquete y el nombre de la clase.
- La notación asterisco (*) después del nombre de un paquete en un **import** indican que todas las clases del paquete deben estar disponibles para el compilador, de modo que éste pueda asegurarse de que las clases utilizan de manera correcta. Esto permite a los programadores utilizar el nombre corto de cualquier clase desde el paquete en el programa.
- Cada instancia (objeto) de una clase contiene una copia de cada variable de instancia. Las variables de instancia se declaran en el cuerpo de la definición de la clase, pero no en el cuerpo de cualquier método de la definición de la clase. Un beneficio importante de las variables de instancia es que sus identificadores pueden utilizarse a través de la definición de la clase (es decir, en todos sus métodos).
- De nuevo, durante la ejecución del applet se llama al método **init**. Por lo general, este método inicializa las variables de instancia del applet y realiza cualquiera de las tareas que necesitan llevarse a cabo una vez, al inicio de la ejecución del applet.
- El método **Double.parseDouble** (un método estático de la clase **Double**) convierte su argumento **String** en un valor de punto flotante. La clase **Double** es parte del paquete **java.lang**.

TERMINOLOGÍA

!= “no es igual que”	clase	diálogo de entrada
< “es menor que”	clase definida por el programador	diálogo de mensaje
<= “es menor o igual que”	clase definida por el usuario	división entera
== “es igual que”	clase Integer	documentar un programa
> “es mayor que”	clase JOptionPane	entero (int)
>= “es mayor o igual que”	clase String	error de compilación
aplicación	clase System	error de sintaxis
applet	comentario (//)	error del compilador
argumento de un método	comentario de documentación de	error en tiempo de compilación
asociatividad de derecha a	Java	extensión de archivo .class
izquierda	comentario de una sola línea	extensión de archivo .java
asociatividad de los operadores	comentario de varias líneas	false
barra de título de un diálogo	compilador	forma en línea recta
biblioteca de clases de Java	concatenación de cadenas	herramienta de comando
cadena	cuadro de diálogo	herramienta shell
cadena de caracteres	cuerpo de la definición de un	identificador
cadena vacía (“”)	método	indicador
carácter de escape diagonal	cuerpo de la definición de una clase	indicador de MS-DOS
invertida (\)	cursor del ratón	instrucción
carácter de nueva línea (\n)	declaración	instrucción de asignación
caracteres blancos	definición de una clase	instrucción import

interfaz de programación de aplicaciones (API) de Java	lista separada por comas	operadores de igualdad
interfaz gráfica de usuario (GUI)	literal	operadores de relación
intérprete	llaves (<code>{ y }</code>)	operando
intérprete <code>java</code>	memoria	palabra reservada <code>class</code>
Java	mensaje	palabra reservada <code>public</code>
Java 2 Software Development Kit (J2SDK)	método	palabra reservada <code>void</code>
<code>JOptionPane.</code>	método <code>main</code>	palabras reservadas
<code>ERROR_MESSAGE</code>	método <code>parseInt</code> de la clase	paquete
<code>JOptionPane.</code>	<code>Integer</code>	paquete <code>java.lang</code>
<code>INFORMATION_MESSAGE</code>	método <code>static</code>	paquete <code>java.swing</code>
<code>JOptionPane.</code>	método <code>System.exit</code>	paréntesis ()
<code>PLAIN_MESSAGE</code>	método <code>System.out.print</code>	paréntesis anidados
<code>JOptionPane.</code>	método <code>System.out.println</code>	precedencia
<code>QUESTION_MESSAGE</code>	navegador Internet Explorer de Microsoft	puntero del ratón
<code>JOptionPane.</code>	navegador Netscape Communicator	reglas de precedencia de operadores
<code>showInputDialog</code>	nombre de una clase	secuencia de escape
<code>JOptionPane.</code>	nombre de variable	sensible a mayúsculas y minúsculas
<code>showMessageDialog</code>	objeto	<code>System.out</code>
<code>JOptionPane.</code>	objeto de salida estándar	terminador de instrucción (;)
<code>WARNING_MESSAGE</code>	operador	terminador de instrucción punto y coma (;)
la llave derecha } termina el cuerpo de un método	operador binario	tipo de dato primitivo
la llave derecha } termina el cuerpo de una clase	operador de asignación (=)	tipo primitivo <code>int</code>
la llave izquierda { comienza el cuerpo de un método	operador de concatenación de cadenas (+)	<code>true</code>
la llave izquierda { comienza el cuerpo de una clase	operador de división (/)	ubicación de memoria
	operador de multiplicación (*)	valor de variable
	operador de suma (+)	variable
	operador de sustracción (-)	ventana de comando
	operador módulo (%)	

ERRORES COMUNES DE PROGRAMACIÓN

- 24.1** Los errores como la división entre cero ocurren durante la ejecución del programa, de modo que estos errores se llaman errores en tiempo de ejecución o errores de ejecución. Los errores fatales en tiempo de ejecución provocan que los programas terminen de inmediato, sin tener éxito al realizar sus tareas. Los errores no fatales en tiempo de ejecución permiten a los programas completar su ejecución, por lo general con resultados incorrectos.
- 24.2** Olvidar uno de los delimitadores de un comentario de varias líneas, es un error de sintaxis.
- 24.3** Java es sensible a mayúsculas y minúsculas. Por lo general, no utilizar las letras mayúsculas y minúsculas apropiadas para un identificador, es un error de sintaxis.
- 24.4** Para una clase pública, es un error si el nombre de archivo no es idéntico al nombre de la clase tanto en las letras, como en las mayúsculas y las minúsculas. Por lo tanto, también es un error que un archivo contenga dos o más clases públicas.
- 24.5** Es un error no finalizar el nombre de un archivo con la extensión `.java`, si contiene la definición una clase de la aplicación. El compilador de Java no podrá compilar la definición de la clase.
- 24.6** Si las llaves no están en pares coincidentes, el compilador indica un error.
- 24.7** Omitir el punto y coma al final de una instrucción, es un error de sintaxis.
- 24.8** Dividir una instrucción a la mitad de un identificador o de una cadena, es un error de sintaxis.
- 24.9** Olvidar llamar a `System.exit` en una aplicación que despliega una interfaz gráfica, evita que el programa termine de manera apropiada. Por lo general, esto provoca que no sea posible introducir comando alguno.
- 24.10** Confundir el operador + utilizado para la concatenación de cadenas con el operador + utilizado para la suma puede provocar resultados extraños. Por ejemplo, al asumir que la variable entera `y` tiene el valor 5, la expresión `"y + 2 = " + y + 2` arroja como resultado la cadena `"y + 2 = 52"`, no `"y + 2 = 7"`, debido a que el primer valor de `y` se concatena con la cadena `"y + 2 ="`, después el valor 2 se concatena con la cadena más grande `"y + = 5"`. La expresión `"y + 2 =" + (y + 2)` produce el resultado deseado.

- 24.11** Colocar caracteres adicionales tales como comas (,) entre los componentes de la etiqueta `<applet>` puede provocar que el `appletviewer` o el navegador produzcan un mensaje de error que indica un `MissingResourceException` al cargar el applet.
- 24.12** Olvidar la etiqueta `</applet>` provoca la carga incorrecta del applet dentro del `appletviewer` o el navegador.
- 24.13** Ejecutar el `appletviewer` con un nombre de archivo que no termina con `.html` o `.htm` provoca un error que evita que el `appletviewer` cargue su applet para ejecución.
- 24.14** Asumir que una instrucción `import` para un paquete completo (por ejemplo, `java.awt.*`) también importa las clases de los subdirectorios de dicho paquete (por ejemplo, `java.awt.event.*`), provoca errores de sintaxis para las clases de los subdirectorios. Debe haber un `import` separado para cada paquete del que se utilizan las clases.
- 24.15** Si las llaves no se encuentran como pares coincidentes, el compilador indica un error de sintaxis.
- 24.16** Utilizar una variable local no inicializada, es un error de sintaxis. A todas las variables locales se les debe asignar un valor antes de intentar utilizar el valor de dicha variable.
- 24.17** Proporcionar un ancho negativo o una altura negativa como argumentos del método `drawRect` de `Graphics`, es un error lógico. El rectángulo no se desplegará y no indicará error alguno.
- 24.18** Proporcionar dos puntos (es decir, pares de coordenadas *x* y *y*) como argumentos del método `drawRect` de `Graphics`, es un error lógico. El tercer argumento debe ser el ancho en pixeles y el cuarto argumento debe ser la altura en pixeles del rectángulo a dibujar.
- 24.19** Por lo general, proporcionar argumentos al método `drawRect` de `Graphics` que provoquen que el rectángulo se dibuje fuera del área visible del applet (es decir, el ancho y la altura del applet como se especifica el documento HTML que hace referencia al applet), es un error lógico. Aumente el ancho y la altura del applet en el documento HTML, o pase los argumentos al método `drawRect` que provoca que el rectángulo se dibuje dentro del área visible del applet.

TIPS PARA PREVENIR ERRORES

- 24.1** Siempre pruebe los programas en Java en todos los sistemas en los que deseé ejecutarlos.
- 24.2** Cuando el compilador reporta un error de sintaxis, el error podría no estar en la línea que indica el mensaje de error. Primero, verifique la línea en donde se reporta el error. Si la línea no contiene errores de sintaxis, verifique las líneas anteriores del programa.
- 24.3** Si el comando `appletviewer` no funciona y/o el sistema indica que el comando `appletviewer` no se encuentra, la variable de ambiente `PATH` podría no estar definida apropiadamente en su computadora. Revise las direcciones de instalación para el Java 2 Software Development Kit para asegurarse de que la variable de ambiente está correctamente definida para su sistema (en algunas computadoras, podría ser necesario reiniciar el equipo después de definir la variable de ambiente `PATH`).
- 24.4** El mensaje de error del compilador “Public class *ClassName* must be defined in a file called *ClassName.java*” indica que 1) el nombre del archivo no coincide exactamente con el nombre de la clase `public` en el archivo (incluidas todas las letras mayúsculas y minúsculas), o 2) que usted escribió el nombre de la clase de manera incorrecta cuando compiló la clase (el nombre debe deletrearse con las letras mayúsculas y minúsculas apropiadas).
- 24.5** Si usted recibe un mensaje de error `MissingResourceException` durante la carga de un applet dentro del `appletviewer` o del navegador, verifique cuidadosamente la etiqueta `<applet>` en el archivo HTML para ver si hay de errores de sintaxis. Compare su archivo HTML con el archivo de la figura 24.15 para confirmar una sintaxis adecuada.

BUENAS PRÁCTICAS DE PROGRAMACIÓN

- 24.1** Escriba sus programas en Java de manera sencilla y directa. A esto en ocasiones se le llama KIS (“keep it simple”, “manténgalo simple”. No deshaga el lenguaje, intentando usos extraños).
- 24.2** Lea la documentación para la versión de Java que va a utilizar. Consulte esta documentación con frecuencia para asegurarse de que conoce la rica colección de características de Java y de que utiliza correctamente estas características.
- 24.3** Su computadora y su compilador son buenos maestros. Si después de leer cuidadosamente el manual de la documentación de Java no está seguro de la manera en que funciona una característica de Java, experimente y vea qué sucede. Estudie cada mensaje de error o de advertencia que obtenga cuando compile sus programas, y corríjalos para eliminar dichos mensajes.

- 24.4** Por convención, usted siempre debe comenzar el nombre de una clase con la primera letra en mayúscula.
- 24.5** Cuando lea un programa en Java, busque identificadores que comiencen con la primera letra en mayúscula. Por lo general, éstos representan clases de Java.
- 24.6** Cada vez que introduzca una llave izquierda de apertura, {, en su programa, introduzca inmediatamente la llave derecha de cierre, }, y vuelva a colocar el indicador entre las llaves para comenzar a introducir el cuerpo del programa. Esto ayuda a evitar que falten llaves.
- 24.7** Sangre el cuerpo entero de cada definición de clase un “nivel” entre la llave izquierda, {, y la llave derecha, }, que define el cuerpo de la clase. Esto enfatiza la estructura de la definición de la clase, y ayuda a que las definiciones de clases sean más fáciles de leer.
- 24.8** Establezca una convención para el tamaño del sangrado que prefiera, y entonces aplique de manera uniforme dicha convención. Puede utilizar la tecla tab para crear el sangrado, aunque tab podría variar entre editores. Le recomendamos el uso de tabuladores de 1/4 de pulgada o (preferiblemente) tres espacios para formar un nivel de sangrado.
- 24.9** Sangre por completo el cuerpo de cada definición de método un “nivel” entre la llave izquierda, {, y la llave derecha, }. Esto hace que la estructura del método resalte, y ayuda a que la definición del método sea más fácil de leer.
- 24.10** Coloque un espacio después de cada coma (,) en una lista de argumentos, para hacer más legibles los programas.
- 24.11** Elegir nombres de variables significativas (descriptivas) ayuda a un programa a estar “autodocumentado” (es decir, se vuelve más sencillo comprender un programa sólo con leerlo, y no es necesario tener que leer los manuales o utilizar comentarios en exceso).
- 24.12** Por convención, los identificadores de nombres de variables comienzan con una letra minúscula. Así como con los nombres de las clases, cada palabra del nombre después de la primera, debe comenzar con una letra mayúscula. Por ejemplo, el identificador **primerNúmero** tiene una letra mayúscula **N** en la segunda palabra **Número**.
- 24.13** Algunos programadores prefieren declarar cada variable en una línea aparte. Este formato permite insertar fácilmente un comentario descriptivo después de cada declaración.
- 24.14** Coloque espacios de cualquier lado de un operador binario. Esto hace que el operador sobresalga y hace al programa más legible.
- 24.15** Investigue cuidadosamente las capacidades de cualquier clase en la documentación del API de Java, antes de heredar a una subclase. Esto ayuda a asegurar que el programador no redefine por descuido una capacidad que ya está proporcionada.
- 24.16** Siempre pruebe el applet en el **appletviewer** y asegúrese de que se ejecuta correctamente, antes de cargar el applet en un navegador de la World Wide Web. Con frecuencia, los navegadores guardan una copia de un applet en la memoria hasta que termina la sesión actual de navegación (es decir, hasta que se cierran todas las ventanas del navegador). Por lo tanto, si usted modifica un applet, recomílelo, y luego recargue el applet en el navegador; es probable que no vea los cambios, debido a que el navegador aún está ejecutando la versión original del applet. Cierre todas las ventanas de su navegador para eliminar de la memoria la versión anterior del applet. Abra una nueva ventana del navegador y cargue el applet para ver los cambios.
- 24.17** Inicializar las variables de instancia en lugar de confiar en la inicialización automática, mejora la legibilidad del programa.

TIP DE RENDIMIENTO

- 24.1** Los intérpretes tienen una ventaja sobre los compiladores en el mundo de Java, a saber, que un programa interpretado puede comenzar su ejecución de inmediato, tan pronto como se descarga en la máquina del cliente, mientras que un programa a compilarse primero debe sufrir un retraso potencialmente largo mientras el programa se compila antes de que pueda ejecutarse.

TIPS DE PORTABILIDAD

- 24.1** Aunque es más fácil escribir programas portables en Java que en la mayoría de los demás lenguajes de programación, existen diferencias entre los compiladores, los intérpretes y las computadoras que pueden hacer de la portabilidad una meta difícil de alcanzar. El simple hecho de escribir programas en Java, no garantiza la portabilidad. Ocasionalmente el programador necesitará lidiar directamente con las variaciones entre los compiladores y las computadoras.
- 24.2** Verifique sus applets en todos los navegadores utilizados por la gente que ve su applet. Esto le ayudará a asegurar que la gente que vea su applet experimente la funcionalidad que usted espera. [Nota: Una meta del plug-in de Java (que explicaremos posteriormente) es proporcionar la ejecución consistente de un applet en diferentes navegadores.]

OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 24.1 Evite utilizar identificadores que contengan signos de moneda (\$), ya que con frecuencia el compilador los utiliza para crear nombres de identificadores.
- 24.2 Si su navegador Web no soporta Java 2, la mayoría de los applets de este libro no se ejecutarán en su navegador. Esto se debe a que la mayoría de los applets de este libro utilizan las características que son nuevas en Java 2, o a que no se proporcionan con los navegadores que soportan Java 1.1.
- 24.3 Por lo general, cada applet debe tener un tamaño menor a 640 pixeles de ancho y 480 pixeles de alto (la mayoría de las pantallas de computadora soportan estas dimensiones de ancho y altura mínimas).
- 24.4 El compilador de Java no necesita instrucciones **import** en un archivo de código fuente de Java si el nombre completo de la clase, es decir, el nombre completo del paquete y el nombre de la clase (por ejemplo, **java.awt.Graphics**), se especifica cada vez que se utiliza el nombre de la clase en el código fuente.
- 24.5 El compilador no carga cada clase en un paquete cuando encuentra una instrucción **import** que utiliza la notación * (por ejemplo, **javax.swing.***) para indicar que se utilizan diversas clases del paquete dentro del programa. El compilador busca en el paquete solamente las clases que utiliza el programa.
- 24.6 Muchos directorios de paquetes tienen subdirectorios. Por ejemplo, el directorio del paquete **java.awt** contiene el subdirectorio **event** para el paquete **java.awt.event**. Cuando el compilador encuentra una instrucción **import** que utiliza la notación * (por ejemplo, **java.awt.***) para indicar que se utilizan distintas clases del paquete dentro del programa, el compilador no busca el subdirectorio **event**. Esto significa que usted no puede definir un **import** de **java.*** para buscar las clases de todos los paquetes.
- 24.7 Cuando utilice instrucciones **import**, debe especificar instrucciones **import** separadas para cada paquete utilizado en el programa.
- 24.8 El orden en que se definen los métodos en la definición de una clase no tiene efecto en cuanto al orden en el que se llaman en tiempo de ejecución.
- 24.9 Una pista para ayudarle a determinar si un identificador es una variable o una referencia es el tipo de dato de la variable. Por convención, todas las clases en Java comienzan con una letra mayúscula. Por lo tanto, si el tipo de dato comienza con una letra mayúscula, por lo general usted puede asumir que el identificador es una referencia a un objeto del tipo declarado (por ejemplo, **Graphics g** indica que **g** es una referencia a un objeto **Graphics**).
- 24.10 Para cada tipo de dato primitivo (tal como un **int** o un **double**) existe una clase correspondiente (tal como **Integer** o **Double**) en el paquete **java.lang**. Estas clases (por lo general conocidas como envolturas de tipo) proporcionan métodos para procesar valores de tipos de datos primitivos (tales como convertir un **String** a un valor de tipo de dato primitivo, o convertir un valor de tipo de dato primitivo a un **String**). Los tipos de datos primitivos no tienen métodos. Por lo tanto, los métodos relacionados con un tipo de dato primitivo se ubican en la clase envolvente de tipo correspondiente (es decir, el método **parseDouble** que convierte un **String** a un valor **double** se localiza en la clase **Double**).
- 24.11 Las únicas instrucciones que deben colocarse en el método **init** del applet son aquellas que se relacionan directamente con la inicialización única de las variables de instancia del applet. Los resultados del applet deben desplegarse a través de otros métodos de la clase del applet. Los resultados que involucran el dibujo deben desplegarse desde el método **paint** del applet.
- 24.12 Las únicas instrucciones que deben colocarse en el método **paint** del applet son aquellas que se relacionan de manera directa con el dibujo (es decir, las llamadas a los métodos de la clase **Graphics**) y con la lógica del dibujo. Por lo general, los cuadros de diálogo no deben desplegarse desde el método **paint** del applet.

EJERCICIOS DE AUTOEVALUACIÓN

- 24.1 Complete los espacios en blanco:
 - a) _____ comienza un comentario de una sola línea.
 - b) La clase _____ despliega diálogos de mensaje y diálogos de entrada.
 - c) Las _____ están reservadas para el uso de Java.
 - d) Las aplicaciones Java comienzan su ejecución en el método _____.
 - e) Los métodos _____ y _____ despliegan información en la ventana de comando.
 - f) Siempre se llama a un método _____ usando el nombre de la clase seguido por un punto (.) y por el nombre de su método.
- 24.2 Diga si es *verdadera* o *falsa* cada una de las siguientes frases. Si es *falsa*, explique por qué.
 - a) Los comentarios provocan que la computadora imprima en la pantalla el texto que se encuentra después de //, cuando se ejecuta el programa.
 - b) Al declararse, todas las variables deben tener un tipo.

- c) Java considera que las variables `numero` y `NuMero` son idénticas.
 - d) El método `Integer.parseInt` convierte un entero a una `String`.
- 24.3** Escriba instrucciones en Java para llevar a cabo cada una de las siguientes tareas:
- a) Declare las variables `c`, `estaEsUnaVariable`, `q76354` y `número` de tipo `int`.
 - b) Despliegue un diálogo que solicite al usuario que introduzca un entero.
 - c) Convierta una `String` a un entero y almacene el valor en la variable entera `edad`. Asuma que la cadena se almacena en `valorCadena`.
 - d) Si la variable `numero` no es igual que 7, despliegue “**La variable numero no es igual que 7**” en un diálogo de mensaje. [Pista: Utilice una versión del diálogo de mensaje que requiere dos argumentos.]
 - e) Imprima el mensaje “**Este es un programa en Java**” en una línea dentro de la ventana de comandos.
 - f) Imprima el mensaje “**Este es un programa en Java**” en dos líneas en la ventana de comandos, en donde la primera línea termina con `programa`. Utilice una sola instrucción.

- 24.4** Identifique y corrija los errores en la siguiente instrucción:

```
if( c=> 7 )
    JOptionPane.showMessageDialog( null,
        "c es igual o mayor que 7");
```

- 24.5** Complete los espacios en blanco.
- a) La clase _____ proporciona métodos para dibujar.
 - b) Los applets de Java comienzan la ejecución con una serie de tres llamadas a los métodos: _____, _____ y _____.
 - c) Los métodos _____ y _____ despliegan líneas y rectángulos.
 - d) La palabra reservada _____ se utiliza para indicar que una nueva clase es una subclase de una clase existente.
 - e) Todo applet de Java debe extenderse a partir de la clase _____ o de la clase _____.
 - f) Una definición de clase describe los _____ y los _____ de un objeto.
 - g) Los ocho tipos de datos primitivos de Java son: _____, _____, _____, _____, _____, _____, _____ y _____.
- 24.6** Diga si es *verdadera* o *falsa* cada uno de las siguientes frases. Si es *falsa*, explique por qué.
- a) El método `drawRect` requiere cuatro argumentos que especifiquen dos puntos en el applet, para dibujar un rectángulo.
 - b) El método `drawLine` requiere cuatro argumentos que especifiquen dos puntos en el applet, para dibujar una línea.
 - c) El tipo `Double` es un tipo de dato primitivo.
 - d) El tipo de dato `int` se utiliza para declarar un número de punto flotante.
 - e) El método `Double.parseDouble` convierte una `String` a un valor primitivo `double`.

- 24.7** Escriba las instrucciones Java para llevar a cabo cada una de las siguientes tareas:
- a) Despliegue un diálogo que pida al usuario que introduzca un número de punto flotante.
 - b) Convierta una `String` a un número de punto flotante y almacene el valor convertido en la variable `double edad`. Asuma que la cadena se almacena en `valorCadena`.
 - c) Dibuje el mensaje “**Este es un programa en Java**” en una línea de un applet (asuma que usted define esta instrucción en el método `paint` del applet) en la posición `(10, 10)`.
 - d) Dibuje el mensaje “**Este es un programa en Java**” en dos líneas de un applet (asuma que estas instrucciones se definen en el método `paint` del applet) que inician en la posición `(10, 10)`, y en donde la primera línea termina con `programa`. Haga que las dos líneas comiencen en la misma coordenada `x`.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

- 24.1** a) //. b) `JOptionPane`. c) Palabras reservadas. d) `main`. e) `System.out.print` y `System.out.println`. f) Estático.
- 24.2** a) Falso. Los comentarios no provocan la ejecución de acción alguna durante la ejecución del programa. Se utilizan para documentar los programas y mejorar su legibilidad.
- b) Verdadero.
- c) Falso. Java es sensible a mayúsculas y a minúsculas, de modo que las variables son distintas.
- d) Falso. El método `Integer.parseInt` convierte una cadena a un valor entero (`int`).

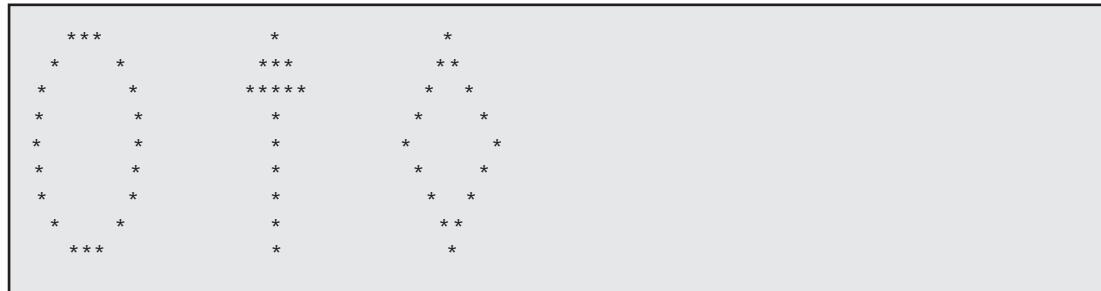
- 24.3** a) `int c, estaEsUnaVariable, q76354, numero;`
 b) `valor = JOptionPane.showInputDialog("Introduzca un entero");`
 c) `edad = Integer.parseInt(valorCadena);`
 d) `if (numero != 7)`
 `JOptionPane.showMessageDialog(null,`
 `"La variable numero no es igual a 7");`
 e) `System.out.println("Este es un programa en Java");`
 f) `System.out.println("Este es un programa\n en Java");`
- 24.4** Error: el operador relacional `=>` es incorrecto.
 Corrección: modifique `=>` por `<=`.
- 24.5** a) `Graphics`. b) `init`, `start` y `paint`. c) `drawLine` y `drawRect`. d) `extends` e) `JApplet`, `Applet`.
 f) Atributos y comportamientos. g) `byte`, `short`, `int`, `float`, `double`, `char` y `boolean`.
- 24.6** a) Falso. El método `drawRect` requiere cuatro argumentos, dos que especifiquen la esquina superior izquierda del rectángulo y dos que especifiquen el ancho y la altura.
 b) Verdadero.
 c) Falso. El tipo `Double` es una clase dentro del paquete `java.lang`. Recuerde que, por lo general, los nombres que comienzan con una letra mayúscula son nombres de clases.
 d) Falso. El tipo de dato `double` o el tipo de dato `float` pueden utilizarse para declarar un número de punto flotante. El tipo de dato `int` se utiliza para declarar enteros.
 e) Verdadero.
- 24.7** a) `valor = JOptionPane.showInputDialog(`
 `"Introduzca un número de punto flotante");`
 b) `edad = Double.parseDouble(valorCadena);`
 c) `g.drawString("Este es un programa en Java", 10, 10);`
 d) `g.drawString("Este es un programa", 10, 10);`
 `g.drawString("en Java", 10, 25);`

EJERCICIOS

- 24.8** Complete los espacios en blanco.
 a) Los _____ se utilizan para documentar un programa y mejorar su legibilidad.
 b) Un diálogo de entrada capaz de recibir la entrada del usuario se despliega con el método _____ de la clase _____.
- 24.9** Escriba una instrucción en Java que lleve a cabo cada una de las siguientes tareas:
 a) Despliegue el mensaje **"Introduzca dos números"** por medio de la clase `JOptionPane`.
 b) Asigne el producto de las variables `b` y `c` a la variable `a`.
 c) Indique que un programa realiza un cálculo de nómina (es decir, utilice texto que ayude a documentar el programa).
- 24.10** ¿Qué se despliega en el diálogo de mensaje cuando se ejecutan cada una de las siguientes instrucciones de Java?
 Asuma que `x = 2` y `y = 3`.
 a) `JOptionPane.showMessageDialog(null, "x = " + x);`
 b) `JOptionPane.showMessageDialog(null,`
 `"El valor de x + x es " + (x + x));`
 c) `JOptionPane.showMessageDialog(null, "x = ");`
 d) `JOptionPane.showMessageDialog(null,`
 `(x + y) + " = " + (y + x));`
- 24.11** Escriba una aplicación que solicite al usuario que introduzca dos números, que obtenga dos números del usuario y que imprima la suma, el producto, la diferencia y el cociente de los dos números. Utilice las técnicas mostradas en la figura 24.7.
- 24.12** Escriba una aplicación que solicite al usuario que introduzca dos enteros, que obtenga los números del usuario y que despliegue el número más grande seguido por las palabras **"es mayor"** dentro de un diálogo de mensaje de información. Si los números son iguales, que imprima el mensaje **"Estos números son iguales"**. Utilice las técnicas mostradas en la figura 24.7.
- 24.13** Escriba una aplicación que introduzca tres enteros del usuario y que despliegue la suma, el promedio, el producto, el más pequeño y el más grande de estos números dentro de un diálogo de mensaje de información. Utilice las téc-

nicas para GUI mostradas en la figura 24.7. [Nota: El cálculo del promedio en este ejercicio debe ser una representación entera del promedio. Así, si la suma de los valores es 7, el promedio será 2 y no 2.333...]

- 24.14** Escriba una aplicación que introduzca el radio de un círculo por parte del usuario y que imprima el diámetro, la circunferencia y el área del círculo. Utilice el valor constante `3.14159` para π . Utilice las técnicas de GUI mostradas en la figura 24.7. [Nota: Podría utilizar también la constante predefinida `Math.PI` para el valor de π . Esta constante es más precisa que el valor `3.14159`. La clase `Math` está definida dentro del paquete `java.lang`, de modo que usted no necesita importarla.] Utilice las siguientes fórmulas (r es el radio): $diámetro = 2r$, $circunferencia = 2\pi r$, $área = \pi r^2$.
- 24.15** Escriba una aplicación que despliegue en la ventana de comando una caja, una elipse, una flecha y un rombo mediante el uso de asteriscos (*), de la siguiente forma:



- 24.16** Modifique el programa que creó en el ejercicio 24.15 para desplegar las formas dentro del diálogo `JOptionPane.PLAIN_MESSAGE`.
- 24.17** Escriba un programa que lea el nombre y el apellido del usuario como dos entradas separadas, y que concatene el nombre y el apellido separados por un espacio. Despliegue el nombre concatenado dentro de un diálogo de mensaje.

25

Más allá de C y C++: Operadores, métodos y arreglos en Java

Objetivos

- Comprender cómo se utilizan los tipos primitivos y los operadores lógicos en Java.
- Introducir los métodos matemáticos comunes disponibles en la API de Java.
- Crear nuevos métodos.
- Comprender los mecanismos utilizados para pasar información entre métodos.
- Introducir técnicas de simulación, utilizando generación de números aleatorios.
- Comprender los objetos de arreglos en Java.
- Comprender cómo escribir y utilizar métodos que se invocan a sí mismos.



La forma siempre sigue a la función.

Louis Henri Sullivan

E pluribus unum.

(Uno compuesto por muchos.)

Virgilio

¡Oh! Volvió a llamar ayer, ofreciéndome volver.

William Shakespeare, *Ricardo II*

Llámame Ismael.

Herman Melville, *Moby Dick*

Cuando me llames así, sonríe.

Owen Wister

Plan general	
25.1	Introducción
25.2	Tipos de datos primitivos y palabras reservadas
25.3	Operadores lógicos
25.4	Definiciones de métodos
25.5	Paquetes de la API de Java
25.6	Generación de números aleatorios
25.7	Ejemplo: Un juego de azar
25.8	Métodos de la clase JApplet
25.9	Declaración y asignación de arreglos
25.10	Ejemplos del uso de arreglos
25.11	Referencias y parámetros de referencias
25.12	Arreglos con múltiples subíndices
<i>Resumen • Terminología • Errores comunes de programación • Tips para prevenir errores • Buenas prácticas de programación • Tips de rendimiento • Tip de portabilidad • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios</i>	

25.1 Introducción

En este capítulo presentamos algunas diferencias clave entre Java, C y C++. Comenzamos presentando tipos de datos primitivos y palabras reservadas de Java. Despues explicamos los operadores lógicos y métodos, así como los paquetes que comprenden la *Interfaz de programación de aplicaciones (API)* de Java.

En el capítulo 5 escribimos un simulador para jugar el juego de craps. En la sección 25.7 retomamos este ejemplo, donde agregamos una *interfaz gráfica de usuario (GUI)* y explicamos cómo generar números aleatorios en Java. Finalizamos el capítulo con una explicación sobre arreglos en Java, y cómo mejoran los arreglos en C y C++.

25.2 Tipos de datos primitivos y palabras reservadas

La tabla de la figura 25.1 lista los tipos de datos primitivos en Java. Los tipos primitivos son bloques de construcción para tipos más complicados. Como sus lenguajes predecesores C y C++, Java requiere que todas las variables tengan un tipo antes de que puedan utilizarse en un programa. Por esta razón, Java se conoce como un *lenguaje fuertemente basado en tipos*.

Tipo	Tamaño en bits	Valores	Estándar
booleano		verdadero o falso	
char	16	'\u0000' a '\uFFFF'	(conjunto de caracteres de ISO Unicode)
byte	8	-128 a +127	
short	16	-32,768 a +32,767	
int	32	-2,147,483,648 a +2,147,483,647	
long	64	-9,223,372,036,854,775,808 a +9,223,372,036,854,775,807	
float	32	-3.40292347E+38 a +3.40292347E+38	(punto flotante IEEE 754)
double	64	-1.79769313486231570E+308 a +1.79769313486231570E+308	(punto flotante IEEE 754)

Figura 25.1 Los tipos de datos primitivos en Java.

A diferencia de C y de C++, los tipos primitivos en Java son portables a través de todas las plataformas de cómputo que soportan Java. Ésta y muchas otras características de portabilidad de Java permiten a los programadores escribir programas una vez, sin conocer la plataforma de cómputo que ejecutará el programa. En ocasiones, a esto se le conoce como “WORA” (Write Once Run Anywhere; Escríbelo una vez, ejecútalo en donde sea).

En programas en C y C++, los programadores con frecuencia tenían que escribir versiones separadas de sus programas para que los soportaran diferentes plataformas, ya que no se garantizaba que los tipos de datos primitivos fueran idénticos de computadora a computadora. Por ejemplo, un valor **int** en una máquina podía representarse con 16 bits (2 bytes) de memoria, y en otra computadora con 32 bits (4 bytes) de memoria. En Java, los valores **int** siempre son de 32 bits (4 bytes).

Tip de portabilidad 25.1



Todos los tipos de datos primitivos en Java son portables, a través de todas las plataformas que soportan Java.

Cada tipo de dato de la tabla se lista con su tamaño en bits (hay 8 bits por un byte), y su rango de valores. Los diseñadores de Java quieren un máximo de portabilidad, por lo que eligieron utilizar estándares internacionalmente reconocidos para formatos de caracteres (Unicode) y para números de punto flotante (IEEE 754).

Siempre que en una clase se declaran instancias de variables de tipos de datos primitivos, se asignan valores predeterminados, a menos que el programador especifique lo contrario. A las variables de tipo **char**, **byte**, **short**, **int**, **long**, **float** y **double** se les da el valor de 0 de manera predeterminada. A las variables de tipo **boolean** se les da de manera predeterminada el valor de **false**.

Cada una de las palabras **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**, son palabras reservadas de Java. Estas palabras están reservadas para el lenguaje, para implementar diversas características, como los tipos de datos primitivos. Las palabras reservadas no pueden utilizarse como identificadores, tal como nombres de variables. En la figura 25.2 aparece una lista completa de las palabras reservadas de Java.

Error común de programación 25.1



Utilizar una palabra reservada como identificador, es un error de sintaxis.

25.3 Operadores lógicos

Java proporciona operadores lógicos que pueden utilizarse para formar condiciones complejas que controlen estructuras mediante la combinación de condiciones simples. Los operadores lógicos son **&&** (AND lógico), **&** (AND lógico booleano), **||** (OR lógico), **|** (OR lógico booleano incluyente), **^** (OR lógico booleano excluyente), y **!** (NOT lógico, también llamado negación lógica). Más adelante consideraremos ejemplos de cada uno de ellos.

Palabras reservadas de Java

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

Palabras que son reservadas, pero que no se utilizan en Java

const **goto**

Figura 25.2 Palabras reservadas de Java.

Suponga que deseamos garantizar en algún punto de un programa que dos condiciones son **true**, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador lógico **&&** de la siguiente manera:

```
if( genero == 1 && edad >= 65 )
    ++MujeresTerceraEdad;
```

Esta instrucción **if** contiene dos condiciones simples. La condición **genero == 1** puede evaluarse, por ejemplo, para determinar si una persona es mujer. La condición **edad >= 65** se evalúa para determinar si una persona es un ciudadano de la tercera edad. Las dos condiciones simples se evalúan primero, ya que las precedencias de **==** y de **>=** son más altas que la precedencia de **&&**. Después, la instrucción **if** considera la condición combinada

```
genero == 1 && edad >= 65
```

Esta condición es **true** si y sólo si ambas condiciones simples son **true**. Por último, si esta condición combinada es **true**, la cuenta de **MujeresTerceraEdad** se incrementa en 1. Si una o ambas condiciones son **false**, el programa evita el incremento y continúa con la instrucción siguiente a la estructura **if**. La condición combinada anterior puede hacerse más legible, agregando paréntesis redundantes:

```
( genero == 1 ) && ( edad >= 65 )
```

La tabla de la figura 25.3 resume el operador **&&**. La tabla muestra las cuatro posibles combinaciones de valores **false** y **true** para la *expresion1* y la *expresion2*. A tales tablas con frecuencia se les conoce como *tablas de verdad*. Java da como resultado **false** o **true** para todas las expresiones que incluyen operadores de relación, de igualdad y/o operadores lógicos.

Ahora consideremos el operador **||** (OR lógico). Suponga que deseamos garantizar que una o ambas condiciones sean **true**, antes de elegir una cierta ruta de ejecución. En este caso, utilizamos el operador **||** como en el siguiente segmento de programa:

```
if( promedioSemestre >= 90 || examenFinal >= 90 )
    System.out.println( "La calificacion del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición **promedioSemestre >= 90** se evalúa para determinar si el estudiante merece una “A” en el curso, debido a un buen desempeño a lo largo del semestre. La condición **examenFinal >= 90** se evalúa para determinar si el estudiante merece una “A” en el curso, debido a un desempeño sobresaliente en el examen final. La instrucción **if** después considera la condición combinada

```
promedioSemestre >= 90 || examenFinal >= 90
```

y otorga al estudiante una “A”, si una o ambas condiciones simples son **true**. Observe que el mensaje “**La calificacion del estudiante es A**”, no se imprime sólo cuando ambas condiciones simples son **false**. La figura 25.4 es una tabla de verdad para el operador lógico OR (**||**).

El operador **&&** tiene una precedencia más alta que el operador **||**. Ambos operadores asocian de izquierda a derecha. Una expresión que contiene los operadores **&&** o **||** se evalúa sólo hasta que se conoce su veracidad o su falsedad. Por lo tanto, la evaluación de la expresión **genero == 1 && edad >= 65** se detendrá inmediatamente si **genero** no es igual que **1** (es decir, la expresión completa es **false**), y continuará si **genero** es igual que **1** (es decir, la expresión completa podría seguir siendo **true**, si la condición **edad >= 65** es

expresion1	expresion2	expresion1 && expresion2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 25.3 Tabla de verdad para el operador **&&** (Y lógico).

expresión1	expresión2	expresión1 expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 25.4 Tabla de verdad para el operador || (OR lógico).

true). Esta característica de desempeño para la evaluación de expresiones lógicas AND y OR se conoce como evaluación en cortocircuito.

Error común de programación 25.2



En expresiones que utilizan el operador `&&`, es posible que una condición (a la que llamaremos condición dependiente) requiera de otra condición para ser `true`, de tal modo que ésta tenga sentido al evaluar la condición dependiente. En este caso, la condición dependiente debe colocarse después de la otra condición, o es posible que ocurra un error.

Tip de rendimiento 25.1



En expresiones que utilizan el operador `&&`, si las condiciones separadas son independientes una de la otra, haga que la condición que más probablemente sea falsa, se encuentre más a la izquierda. En expresiones que utilizan el operador `||`, haga que la condición que más probablemente sea verdadera, se encuentre más a la izquierda. Esto puede reducir el tiempo de ejecución de un programa.

Los operadores *AND lógico booleano* (`&`) y *OR lógico booleano incluyente* (`|`) funcionan de manera idéntica a los operadores lógicos AND y OR, con una excepción: los operadores lógicos booleanos siempre evalúan sus dos operandos (es decir, no hay una evaluación en cortocircuito). Por lo tanto, la expresión

```
genero == 1 & edad >= 65
```

evalúa `edad >= 65` independientemente de si `genero` es igual que 1. Esto es útil si el operando derecho del operador lógico booleano AND, o el operador lógico booleano incluyente OR tiene un *efecto colateral* necesario; una modificación al valor de una variable. Por ejemplo, la expresión

```
cumpleanos == true | ++edad >= 65
```

garantiza que la condición `++edad >= 65` será evaluada. Entonces, la variable `edad` se incrementará en la expresión anterior, independientemente de si la expresión completa es `true` o `false`.

Buena práctica de programación 25.1



Por claridad, evite expresiones con efectos colaterales en las condiciones. Los efectos colaterales pueden parecer convenientes, pero con frecuencia representan más problemas que ventajas.

Una condición que contiene el operador *OR lógico booleano excluyente* (`^`) es `true`, si y sólo si uno de sus operandos resulta en un valor `true` y uno resulta en un valor `false`. Si ambos operandos son `true`, o ambos son `false`, el resultado de la condición completa es `false`. La figura 25.5 es una tabla de verdad para

expresión1	expresión2	expresión1 ^ expresión2
false	false	false
false	true	true
true	false	true
true	true	false

Figura 25.5 Tabla de verdad para el operador lógico booleano excluyente OR (^).

el operador lógico booleano excluyente OR (^). Este operador también garantiza la evaluación de sus dos operandos (es decir, no existe una evaluación de cortocircuito).

Java proporciona el operador ! (negación lógica) para permitir al programador “revertir” el significado de una condición. A diferencia de los operadores lógicos &&, &, ||, | y ^, los cuales combinan dos condiciones (operadores binarios), el operador de negación lógica tiene solamente una condición como operando (operador unario). El operador de negación lógica se coloca antes de una condición para elegir una ruta de ejecución, si la condición original (sin el operador de negación lógica) es **false**, como en el siguiente segmento de programa:

```
if ( ! ( calificacion == valorCentinela ) )
    System.out.println( "La siguiente calificacion es " + calificacion );
```

Los paréntesis alrededor de la condición **calificacion == valorCentinela** son necesarios, ya que el operador de negación lógica tiene una precedencia más alta que el operador de igualdad. La figura 25.6 es una tabla de verdad para el operador de negación lógica.

En la mayoría de los casos, el programador puede evitar el uso de la negación lógica, expresando la condición de manera diferente con un operador de igualdad o de relación adecuado. Por ejemplo, la instrucción anterior puede escribirse de la siguiente manera:

```
if( calificacion != valorCentinela )
    System.out.println( "La siguiente calificacion es " + calificacion );
```

Esta flexibilidad puede ayudar al programador a expresar una condición de una manera más conveniente. La aplicación de la figura 25.7 muestra todos los operadores lógicos y booleanos, produciendo sus tablas de verdad. El programa utiliza la concatenación de cadenas para crear la cadena que se despliega en un **JTextArea**.

En la salida de la figura 25.7, las cadenas “verdadero” y “falso” indican **false** y **true** para los operandos de cada condición. El resultado de la condición aparece como **true** o **false**. Observe que siempre que agrega un valor **boolean** a una **String**, Java agrega la cadena “false” o “true”, basándose en el valor booleano.

expresión	!expresión
false	true
true	false

Figura 25.6 Tabla de verdad para el operador !(NOT lógico).

```
1 // Figura 25.7: OperadoresLogicos.java
2 // Demostración de los operadores lógicos
3 import javax.swing.*;
4
5 public class OperadoresLogicos {
6     public static void main( String args[] )
7     {
8         JTextArea areaSalida = new JTextArea( 17, 20 );
9         JScrollPane desplaza = new JScrollPane( outputArea );
10        String salida = "";
11
12        salida += " AND Logico (&&)" +
13                    "\nfalso && falso: " + ( false && false ) +
14                    "\nfalso && verdadero: " + ( false && true ) +
15                    "\nverdadero && falso: " + ( true && false ) +
16                    "\nverdadero && verdadero: " + ( true && true );
```

Figura 25.7 Demostración de los operadores lógicos. (Parte 1 de 2.)

```

17
18     salida += "\n\n OR Logico (||) " +
19             "\nfalso || falso: " + ( false || false ) +
20             "\nfalso || verdadero: " + ( false || true ) +
21             "\nverdadero || falso: " + ( true || false ) +
22             "\nverdadero || verdadero: " + ( true || true );
23
24     salida += "\n\nAND Logico Booleano (&)" +
25             "\nfalso & falso: " + ( false & false ) +
26             "\nfalso & verdadero: " + ( false & true ) +
27             "\nverdadero & falso: " + ( true & false ) +
28             "\nverdadero & verdadero: " + ( true & true );
29
30     salida += "\n\n OR Logico Booleano Incluyente ()" +
31             "\nfalso | falso: " + ( false | false ) +
32             "\nfalso | verdadero: " + ( false | true ) +
33             "\nverdadero | falso: " + ( true | false ) +
34             "\nverdadero | verdadero: " + ( true | true );
35
36     salida += "\n\nOR Logico Booleano Excluyente (^)" +
37             "\nfalso ^ falso: " + ( false ^ false ) +
38             "\nfalso ^ verdadero: " + ( false ^ true ) +
39             "\nverdadero ^ falso: " + ( true ^ false ) +
40             "\nverdadero ^ verdadero: " + ( true ^ true );
41
42     salida += "\n\nNOT Logico (!)" +
43             "\n!falso: " + ( !false ) +
44             "\n!verdadero: " + ( !true );
45
46     outputArea.setText( salida );
47     JOptionPane.showMessageDialog( null, scroller,
48             "Tablas de verdad", JOptionPane.INFORMATION_MESSAGE );
49     System.exit( 0 );
50 } // fin de main
51 } // fin de la clase OperadoresLogicos

```

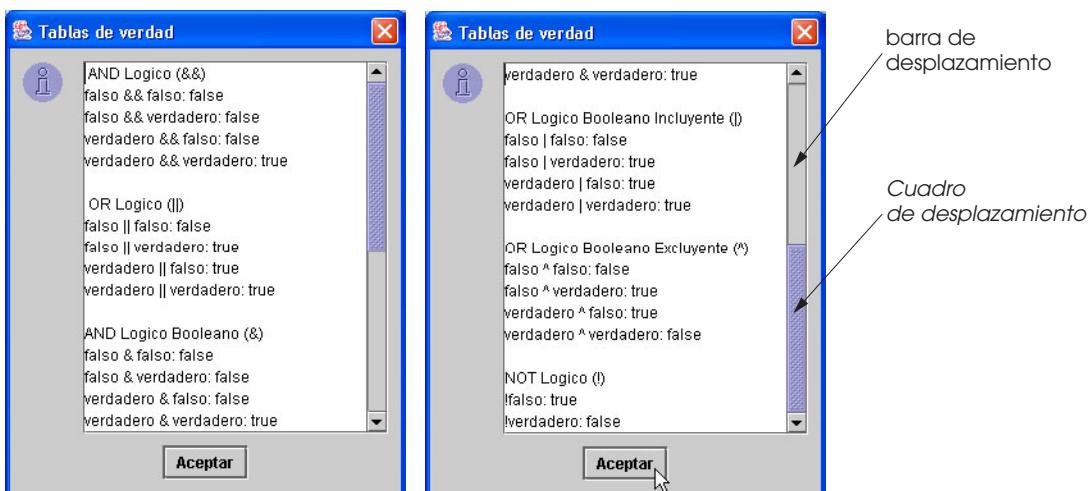


Figura 25.7 Demostración de los operadores lógicos. (Parte 2 de 2.)

La línea 8 del método main

```
JTextArea areaSalida = new JTextArea( 17, 20 );
```

crea un **JTextArea** con 17 filas y 20 columnas. La línea 9

```
JScrollPane desplaza = new JScrollPane( areaSalida );
```

declara la referencia **desplaza** de **JScrollPane** y lo inicializa con un nuevo objeto **JScrollPane**. La clase **JScrollPane** (del paquete **javax.swing**) proporciona un componente GUI con funcionalidad de desplazamiento.

Cuando ejecute esta aplicación, observe la *barra de desplazamiento* del lado derecho de **JTextArea**. Puede hacer clic en las *flechas* superior o inferior de la barra de desplazamiento para desplazarse hacia arriba o hacia abajo a lo largo del texto del **JTextArea**, una línea a la vez. También puede arrastrar el *cuadro de desplazamiento* (también llamado el *pulgár*) hacia arriba o hacia abajo, para desplazarse rápidamente por el texto. Un objeto **JScrollPane** se inicializa con el componente GUI para el que proporcionará la funcionalidad de desplazamiento (en este caso, **areaSalida**). Esto adjunta el componente GUI al **JScrollPane**.

Las líneas 12 a 44 construyen la cadena **salida** que se desplegará en el **areaSalida**. La línea 46 utiliza el método **setText** para remplazar el texto de **areaSalida** con el de la cadena **salida**. Las líneas 47 y 48 despliegan un diálogo de mensaje. El segundo argumento, **desplaza**, indica que el **desplaza** y el **areaSalida** adjunto a él deben desplegarse como el mensaje del diálogo de mensaje.

25.4 Definiciones de métodos

Todos los programas que hemos presentado consisten en una definición de clase que al menos contiene una definición de métodos, llamados métodos API de Java, para realizar sus tareas. Ahora consideraremos cómo es que los programadores escriben sus propios métodos personalizados.

Considere un applet (figura 25.8) que utiliza un método **cuadrado** (invocado desde el método **init** del applet) para calcular los cuadrados de enteros en el rango de 1 a 10.

Cuando el applet comienza su ejecución, se llama primero a su método **init**. La línea 9 declara la referencia **salida** de **String**, y la inicializa con la cadena vacía. Esta **String** contendrá los resultados de elevar al cuadrado los valores de 1 a 10. La línea 11 declara la referencia **areaSalida** de **JTextArea**, y la

```

1 // Figura 25.8: CuadradoEnt.java
2 // Método cuadrado definido por el programador
3 import java.awt.Container;
4 import javax.swing.*;
5
6 public class CuadradoEnt extends JApplet {
7     public void init()
8     {
9         String salida = "";
10
11         JTextArea areaSalida = new JTextArea( 10, 20 );
12
13         // obtiene el área de visualización del componente GUI del applet
14         Container c = getContentPane();
15
16         // adjunta el areaSalida al Contenedor c
17         c.add( areaSalida );
18
19         int resultado;
20

```

Figura 25.8 Uso del método **cuadrado** definido por el programador. (Parte 1 de 2.)

```

21      for ( int x = 1; x <= 10; x++ ) {
22          resultado = cuadrado( x );
23          salida += "El cuadrado de " + x +
24              " es " + resultado + "\n";
25      } // fin de for
26
27      areaSalida.setText( salida );
28  } // fin del método init
29
30  // definición del método cuadrado
31  public int cuadrado( int y )
32  {
33      return y * y;
34  } // fin del método cuadrado
35 } // fin de la clase CuadradoEnt

```

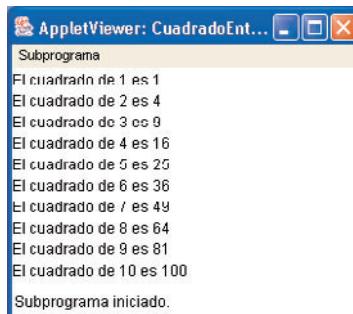


Figura 25.8 Uso del método **cuadrado** definido por el programador. (Parte 2 de 2.)

inicializa con un nuevo objeto **JTextArea** de 10 filas y 20 columnas. La cadena **salida** se desplegará en **areaSalida**.

Este programa es el primero en el que desplegamos un componente GUI en un applet. El área de la pantalla en la que se despliega un **JApplet** tiene un *panel de contenido* en el que los componentes GUI deben adjuntarse, de tal modo que puedan desplegarse en tiempo de ejecución. El panel de contenido es un objeto de la clase *Container* del paquete **java.awt**. Esta clase se importó en la línea 3 para utilizarla en el applet. La línea 14

```
Container c = getContentPane();
```

declara la referencia **c** de **Container**, y la asigna al resultado de una llamada al método **getContentPane**; uno de los muchos métodos que nuestra clase **CuadradoEnt** hereda de la clase **JApplet**. El método **getContentPane** devuelve una referencia al panel de contenido del applet, la cual puede utilizarse para adjuntar componentes GUI, tales como un **JTextArea**, a la interfaz de usuario de la applet.

La línea 17

```
c.add( areaSalida );
```

coloca en el applet el objeto componente de la GUI, **JTextArea**, al que hace referencia **areaSalida**, para que pueda desplegarse cuando el applet se ejecuta. El método **add** de **Container** adjunta un componente GUI al contenedor. Por el momento, sólo podemos adjuntar un componente GUI al panel de contenido del applet, y ese componente GUI automáticamente ocupará toda el área de dibujo del applet en la pantalla (como definieron el **width** y la **height** del applet en pixeles, en el documento HTML del applet). Más adelante explicaremos cómo distribuir varios componentes GUI en un applet.

La línea 19 declara la variable **int, resultado**, en la que se almacena el resultado de calcular cada cuadrado. Las líneas 21 a 25 corresponden a una estructura **for**, en la que cada iteración del ciclo calcula el

cuadrado del valor actual de la variable de control **x**, almacena el valor en **resultado** y concatena el **resultado** al final de **salida**.

El método **cuadrado** se *invoca* o se *llama* en la línea 22, por medio de la instrucción

```
resultado = cuadrado( x );
```

Cuando el control del programa alcanza esta instrucción, el método **cuadrado** (definido en la línea 31) es invocado. De hecho, los () representan el *operador de llamada a métodos*, el cual tiene una precedencia alta. En este punto, el programa hace automáticamente una copia del valor de **x** (el *argumento* de la llamada al método), y el control del programa se transfiere a la primera línea del método **cuadrado**. El método **cuadrado** recibe la copia del valor de **x** en el *parámetro* **y**. Después, **cuadrado** calcula **y * y**. El resultado se pasa de regreso hacia el punto en **init** donde se invocó a **cuadrado**. El valor devuelto se asigna entonces a la variable **resultado**. Las líneas 23 y 24

```
salida += "El cuadrado de " + x +
           " es " + resultado + "\n";
```

concatenan “**El cuadrado de**”, el valor de **x**, “**es**”, el valor de **resultado**, y un carácter de nueva línea al final de **salida**. Este proceso se repite diez veces, por medio de la estructura de repetición **for**. La línea 27

```
areaSalida.setText( salida );
```

utiliza el método **setText** para establecer el texto de **areaSalida** a la **String** **salida**. Observe que las referencias **salida**, **areaSalida**, **c** y la variable **resultado** se declaran como variables locales en **init**, ya que sólo se utilizan en **init**. Las variables deben declararse como variables de instancia, sólo si es necesario utilizarlas en más de un método de la clase, o si sus valores deben guardarse entre llamadas a los métodos de la clase.

La definición del método **cuadrado** (línea 31) muestra que **cuadrado** espera un parámetro entero **y**; éste será el nombre utilizado para manipular el valor pasado a **cuadrado** en el cuerpo del método. La palabra reservada **int** que precede al nombre del método indica que **cuadrado** devuelve un resultado entero. La instrucción **return** de **cuadrado** pasa el resultado del cálculo **y * y** de regreso al método que hizo la llamada. Observe que la definición completa del método se encuentra entre las llaves de la clase **CuadradoEnt**. Todos los métodos deben declararse dentro de una definición de clase.

Buena práctica de programación 25.2



Coloque una línea en blanco entre las definiciones de métodos para separarlos y para mejorar la legibilidad del programa.

Error común de programación 25.3



Definir un método fuera de las llaves correspondientes a la definición de una clase, es un error de sintaxis.

El formato para la definición de un método es

```
tipo del valor de retorno nombre del método (lista de parámetros)
{
  declaraciones e instrucciones
}
```

El *nombre del método* es cualquier identificador válido. El *tipo del valor de retorno* es el tipo de dato del resultado devuelto por el método a quien hizo la llamada. El tipo del valor de retorno **void** indica que un método no devuelve valor alguno. Los métodos pueden devolver, cuando mucho, un valor.

Error común de programación 25.4



Omitir el tipo del valor de retorno en la definición de un método, es un error de sintaxis.

Error común de programación 25.5



Olvidar devolver un valor por parte de un método que se supone debe hacerlo, es un error de sintaxis. Si se especifica un tipo de valor de retorno diferente de void, el método debe contener una instrucción return.

Error común de programación 25.6



Devolver un valor desde un método, cuyo tipo de retorno se declaró como **void**, es un error de sintaxis.

La *lista de parámetros* es una lista separada por comas que contiene las declaraciones de los parámetros recibidos por el método cuando es llamado. En la llamada al método debe haber un argumento para cada parámetro en la definición del método. Los argumentos también deben ser compatibles con el tipo del parámetro. Por ejemplo, un tipo de parámetro **double** podría recibir valores de **7.35, 22, o -0.3546**, pero no "**hola**" (ya que una variable **double** no puede contener un **String**). Si un método no recibe valores, la *lista de parámetros* está vacía (es decir, al nombre del método le sigue un conjunto de paréntesis vacío). Un tipo debe listarse explícitamente para cada parámetro de la lista de un método, u ocurrirá un error de sintaxis.

Error común de programación 25.7



Declarar parámetros del mismo tipo en un método, como **float x, y**, en lugar de **float x, float y**, es un error de sintaxis, ya que se necesitan tipos para cada parámetro de la lista de parámetros.

Error común de programación 25.8



Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de una definición de método, es un error de sintaxis.

Error común de programación 25.9



Redefinir un parámetro de un método como una variable local del método, es un error de sintaxis.

Error común de programación 25.10



Pasar un método a un argumento que no es compatible con el tipo correspondiente al parámetro, es un error de sintaxis.

Buena práctica de programación 25.3



Aunque no es incorrecto hacerlo, en la definición de un método no utilice los mismos nombres para los argumentos pasados a él y para los parámetros correspondientes. Esto ayuda a evitar la ambigüedad.

Las *declaraciones e instrucciones* entre llaves forman el *cuerpo del método*. Al cuerpo del método también se le conoce como *bloque*. Un bloque es una instrucción compuesta que incluye declaraciones. Las variables pueden declararse en cualquier bloque, y los bloques pueden estar anidados. Un método no puede definirse dentro de otro método.

Error común de programación 25.11



Definir un método dentro de otro, es un error de sintaxis.

Buena práctica de programación 25.4



Elegir nombres descriptivos para los métodos y para los parámetros hace que los programas sean más legibles, y ayuda a evitar el uso excesivo de comentarios.

Observación de ingeniería de software 25.1



Por lo general, un método no debe sobrepasar una página. Mejor aún, un método generalmente debe abarcar no más de media página. Independientemente de cuán largo sea un método, debe realizar bien una tarea. Los métodos pequeños promueven la reutilización de software.

Tip para prevenir errores 25.1



Los métodos pequeños son más fáciles de probar, depurar y comprender, que aquellos que son grandes.

Observación de ingeniería de software 25.2



Los programas deben escribirse como colecciones de métodos pequeños. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.



Observación de ingeniería de software 25.3

Es posible que un método que requiere un gran número de parámetros esté realizando demasiadas tareas. Considera dividir el método en métodos más pequeños que realicen tareas separadas. Si es posible, el encabezado del método debe caber en una línea.



Observación de ingeniería de software 25.4

El encabezado de un método y las llamadas a él deben coincidir en número, tipo y orden de parámetros y argumentos.

Existen tres formas para devolver el control al punto en el que se invocó a un método. Si el método no devuelve un resultado, el control se devuelve cuando se alcanza la llave derecha de terminación del método, o ejecutando la instrucción

```
return;
```

Si el método devuelve un resultado, la instrucción

```
return expresión;
```

devuelve el valor de *expresión* a quien hizo la llamada. Cuando se ejecuta una instrucción **return**, el control vuelve inmediatamente al punto en el que se invocó al método.

Observe que el ejemplo de la figura 25.8 en realidad contiene dos definiciones de métodos; **init** (línea 7) y **cuadrado** (línea 31). Recuerde que el método **init** es llamado automáticamente para inicializar el applet. En este ejemplo, el método **init** invoca de manera reiterada al método **cuadrado** para que realice un cálculo, después despliega los resultados en el **JTextArea** que está adjunto al panel de contenido del applet.

Observe la sintaxis utilizada para invocar al método **cuadrado**; sólo utilizamos el nombre del método, seguido por los argumentos de éste entre paréntesis. Por medio de esta sintaxis, a los métodos en una definición de clase se les permite invocar a todos los métodos restantes de la misma definición de clase (existe una excepción, la cual explicaremos en el capítulo 26). Los métodos en la misma definición de clase son los métodos definidos en esa clase y los métodos heredados (los métodos de la clase que la clase actual **extiende (extends)**; en el último ejemplo, **JApplet**). Ahora hemos visto tres formas para llamar a un método; por el nombre mismo del método (como mostramos con **cuadrado(x)**, en este ejemplo), por medio de una referencia a un objeto seguido por el operador punto (**.**) y por el nombre del método [como en **g.drawLine(x1, y1, x2, y2)**], y por medio del nombre de una clase seguido por el nombre del método [como en **Integer.parseInt(stringToConvert)**]. La última sintaxis es sólo para métodos **static** de una clase (los cuales explicaremos con detalle en el capítulo 26).

25.5 Paquetes de la API de Java

Como hemos visto, Java contiene muchas clases predefinidas que están agrupadas en directorios del disco, en categorías de clases relacionadas llamadas paquetes. Juntos, estos paquetes se conocen como la interfaz de programación de aplicaciones de Java (API de Java).

Tipo	Promociones permitidas
double	Ninguna
float	double
long	float o double
int	long, float o double
char	int, long, float o double
short	int, long, float o double
byte	short, int, long, float o double
boolean	Ninguna (en Java, los valores booleanos no se consideran como números)

Figura 25.9 Promociones permitidas para tipos de datos primitivos.

A lo largo del texto, utilizamos las instrucciones **import** para especificar la ubicación de las clases requeridas para compilar un programa en Java. Por ejemplo, para indicarle al compilador que cargue la clase **JApplet** del paquete **javax.swing**, se utiliza la instrucción

```
import javax.swing.JApplet;
```

Una de las grandes fortalezas de Java es el gran número de clases en los paquetes de la API de Java, las cuales pueden reutilizar los programadores, en lugar de “reinventar la rueda”. En este libro practicamos con muchas de estas clases. La figura 25.10 lista alfabéticamente los paquetes de la API de Java, y proporciona una breve descripción de cada uno. Es posible descargar otros paquetes disponibles, desde <http://java.sun.com>. Observa que aún no hemos explicado la mayoría de estos paquetes. Esta tabla se la proporcionamos para darle una idea de la variedad de componentes reutilizables que se encuentran disponibles en la API de Java. Cuando se aprende Java, uno debe invertir tiempo en leer las descripciones de los paquetes y las clases en la documentación de la API de Java.

Paquete	Descripción
java.applet	<i>The Java Applet Package.</i> Este paquete contiene la clase Applet y diversas interfaces que permiten la creación de applets, la interacción de applets con el navegador y con los clips de reproducción de audio. En Java 2, la clase javax.swing.JApplet se utiliza para definir un applet que utiliza los <i>Swing GUI components</i> .
java.awt	<i>The Java Abstract Windowing Toolkit Package.</i> Este paquete contiene las clases e interfaces requeridas para crear y manipular interfaces gráficas de usuario en Java 1.0 y 1.1. En Java 2, estas clases pueden utilizarse, pero con frecuencia, en su lugar se utilizan los <i>Swing GUI components</i> de los paquetes javax.swing .
java.awt.color	<i>The Java Color Space Package.</i> Este paquete contiene clases que soportan espacios de color.
java.awt.datatransfer	<i>The Java Data Transfer Package.</i> Este paquete contiene clases e interfaces que permiten la transferencia de datos entre un programa en Java y el portapapeles de la computadora (un área de almacenamiento temporal para datos).
java.awt.dnd	<i>The Java Drag-and-Drop Package.</i> Este paquete contiene clases e interfaces que proporcionan capacidades para arrastrar y soltar programas.
java.awt.event	<i>The Java Abstract Windowing Toolkit Event Package.</i> Este paquete contiene clases e interfaces que permiten el manejo de eventos para componentes GUI en los paquetes java.awt y javax.swing .
java.awt.font	<i>The Java Font Manipulation Package.</i> Este paquete contiene clases e interfaces para manipular diferentes fuentes.
java.awt.geom	<i>The Java Two-Dimensional Objects Package.</i> Este paquete contiene clases para manipular objetos que representan gráficos bidimensionales.
java.awt.im	<i>The Java Input Method Framework Package.</i> Este paquete contiene clases y una interfaz que soporta entrada en los lenguajes japonés, chino y coreano en un programa en Java.
java.awt.image	<i>The Java Image Packages.</i>
java.awt.image.renderable	Estos paquetes contienen clases e interfaces que permiten el ordenamiento y la manipulación de imágenes en un programa.

Figura 25.10 Paquetes de la API de Java. (Parte 1 de 4.)

Paquete	Descripción
java.awt.print	<i>The Java Printing Package.</i> Este paquete contiene clases e interfaces que soportan la impresión desde programas en Java.
java.beans	<i>The Java Beans Packages.</i>
java.beans.beancontext	Estos paquetes contienen clases e interfaces que permiten al programador crear componentes reutilizables de software.
java.io	<i>The Java Input/Output Package.</i> Este paquete contiene clases que permiten a los programas introducir y desplegar datos.
java.lang	<i>The Java Language Package.</i> Este paquete contiene clases e interfaces requeridas por muchos programas en Java (muchos de cuales explicamos a lo largo del texto), y el compilador lo importa automáticamente hacia todos los programas.
java.lang.ref	<i>The Reference Objects Package.</i> Este paquete contiene clases que permiten la interacción entre un programa en Java y un recolector de basura.
java.lang.reflect	<i>The Java Core Reflection Package.</i> Este paquete contiene clases e interfaces que permiten a un programa descubrir de manera dinámica las variables y los métodos accesibles de una clase durante la ejecución de un programa.
java.math	<i>The Java Arbitrary Precision Math Package.</i> Este paquete contiene clases para realizar aritmética con una precisión arbitraria.
java.net	<i>The Java Networking Package.</i> Este paquete contiene clases que permiten a los programas comunicarse a través de redes.
java.rmi	<i>The Java Remote Method Invocation Packages.</i>
java.rmi.activation	Estos paquetes contienen clases e interfaces que permiten al programador crear programas distribuidos en Java. Al utilizar un método de invocación remoto, un programa puede llamar a un método de un programa separado en la misma computadora, o en una computadora en cualquier parte por medio de Internet.
java.rmi.dgc	
java.rmi.registry	
java.rmi.server	
java.security	<i>The Java Security Packages.</i>
java.security.acl	Estos paquetes contienen clases e interfaces que permiten a un programa en Java encriptar los datos y controlar los privilegios de acceso proporcionados a un programa en Java para efectos de seguridad.
java.security.cert	
java.security.interfaces	
java.security.spec	
java.sql	<i>The Java Database Connectivity Package.</i> Este paquete contiene clases e interfaces que permiten a un programa en Java interactuar con una base de datos.
java.text	<i>The Java Text Package.</i> Este paquete contiene clases e interfaces que permiten a un programa en Java manipular números, fechas, caracteres y cadenas. Este paquete también proporciona muchas de las capacidades de internacionalización de Java para personalizar aplicaciones locales (o de una región geográfica en particular).

Figura 25.10 Paquetes de la API de Java. (Parte 2 de 4.)

Paquete	Descripción
java.util	<i>The Java Utilities Package.</i> Este paquete contiene clases de utilidad e interfaces como: manipulaciones de fecha y hora, capacidades para procesamiento de números aleatorios (Random), almacenamiento y procesamiento de grandes cantidades de datos, romper cadenas en piezas pequeñas llamadas tokens (StringTokenizer), y otras capacidades.
java.util.jar	<i>The Java Utilities JAR and ZIP Packages.</i>
java.util.zip	Estos paquetes contienen clases de utilidad e interfaces que permiten a un programa en Java combinar archivos de Java .class y otros archivos de recursos (como imágenes y audio) en archivos comprimidos llamados <i>archivos Java archive (JAR)</i> o <i>archivos ZIP</i> .
javax.accessibility	<i>The Java Accessibility Package.</i> Este paquete contiene clases e interfaces que permiten a un programa en Java soportar tecnologías para gente con discapacidades; algunos ejemplos son los lectores de pantalla y los magnificadores de pantalla.
java.swing	<i>The Java Swing GUI Components Package.</i> Este paquete contiene clases e interfaces para los componentes Swing GUI de Java que proporcionan soporte para GUIs portables.
javax.swing.border	<i>The Java Swing Borders Package.</i> Este paquete contiene clases y una interfaz para dibujar límites alrededor de áreas en una GUI.
javax.swing.colorchooser	<i>The Java Swing Color Chooser Package.</i> Este paquete contiene clases e interfaces para el diálogo predefinido JColorChooser para elegir colores.
javax.swing.event	<i>The Java Swing Event Package.</i> Este paquete contiene clases e interfaces que permiten la manipulación de eventos para componentes GUI en el paquete javax.swing .
javax.swing.filechooser	<i>The Java Swing File Chooser Package.</i> Este paquete contiene clases e interfaces para el diálogo predefinido JFileChooser para localizar archivos en disco.
javax.swing.plaf	<i>The Java Swing Pluggable-Look-and-Feel Packages.</i>
javax.swing.plaf.basic	Estos paquetes contienen clases y una interfaz que se utilizan para cambiar la apariencia visual de una GUI basada en Swing, entre la apariencia visual de Java, la apariencia visual de Windows de Microsoft y la de UNIX Motif.
javax.swing.plaf.metal	El paquete también soporta el desarrollo de una apariencia visual personalizada para un programa en Java.
javax.swing.plaf.multi	
javax.swing.table	<i>The Java Swing Table Package.</i> Este paquete contiene clases e interfaces para crear y manipular tablas al estilo de hojas de cálculo.
javax.swing.text	<i>The Java Swing Text Package.</i> Este paquete contiene clases e interfaces para manipular texto basado en componentes GUI en Swing.
javax.swing.text.html	<i>The Java Swing HTML Text Packages.</i>
javax.swing.text.html.parser	Estos paquetes contienen una clase que proporciona soporte para construir editores de texto HTML.

Figura 25.10 Paquetes de la API de Java. (Parte 3 de 4.)

Paquete	Descripción
<code>javax.swing.text.rtf</code>	<i>The Java Swing RTF Text Package.</i> Este paquete contiene una clase que proporciona soporte para construir editores que soportan un rico formato de texto.
<code>javax.swing.tree</code>	<i>The Java Swing Tree Package.</i> Este paquete contiene clases e interfaces para crear y manipular árboles de expansión de componentes GUI.
<code>javax.swing.undo</code>	<i>The Java Swing Undo Package.</i> Este paquete contiene clases e interfaces que soportan el proporcionar capacidades de hacer y deshacer a un programa en Java.
<code>org.omg.CORBA</code> <code>org.omg.CORBA.DynAnyPackage</code> <code>org.omg.CORBA.ORBPackage</code> <code>org.omg.CORBA.portable</code> <code>org.omg.CORBA.</code> <code>TypeCodePackage</code> <code>org.omg.CosNaming</code> <code>org.omg.CosNaming.</code> <code>NamingContextPackage</code>	<i>The Object Management Group (OMG) CORBA Packages.</i> Estos paquetes contienen clases e interfaces que implementan APIs CORBA de OMG que permiten a un programa en Java comunicarse con programas escritos en otros lenguajes de programación, de manera similar a cuando se utilizan los paquetes RMI de Java para comunicación entre programas en Java.

Figura 25.10 Paquetes de la API de Java. (Parte 4 de 4.)

25.6 Generación de números aleatorios

Ahora veremos nuevamente la simulación y los juegos (vea el capítulo 5). Como C, Java proporciona al programador los métodos para generar números aleatorios. En esta sección y en la siguiente, desarrollaremos versiones en Java de los programas de juegos que escribimos anteriormente en C.

Como recordará, en C utilizamos la función `rand` para generar números aleatorios. La función `rand` devolvía un valor entero entre 0 y la constante simbólica `RAND_MAX`. Los números aleatorios se generan de manera diferente en Java. La clase `Math` proporciona el método `random`. Considere la siguiente instrucción:

```
double valorAleatorio = Math.random()
```

El método `random` genera un valor `double` mayor o igual que 0.0, pero menor que 1.0. Si `random` realmente produce valores al azar, todo valor mayor o igual que 0.0, pero menor que 1.0, tiene una *probabilidad* igual de ser elegido cada vez que se llama a `random`.

El rango de valores producidos directamente por `random`, con frecuencia es diferente de lo que se necesita en una aplicación específica. Por ejemplo, un programa que simula el tiro de un dado de seis lados requeriría números aleatorios en el rango de 1 a 6. Un programa que al azar predice el siguiente tipo de nave espacial (fuera de cuatro posibilidades) que volará a través del horizonte en un juego de video requeriría enteros aleatorios en el rango de 1 a 4.

Para mostrar `random`, desarrollemos una versión en Java del programa del capítulo 5 que simula 20 tiros de un dado y que imprime el valor de cada tiro. Utilicemos el operador de multiplicación (*) junto con `random` de la siguiente manera

```
(int) ( Math.random() *6 )
```

para producir enteros en el rango de 0 a 5. Recordará que en C utilizamos el operador módulo (%) para *escalar* el valor de retorno de `rand`. Debido a que el método `random` de Java devuelve un valor `double` mayor o igual que 0.0, pero menor que 1.0, debemos multiplicar el número aleatorio por un factor de escala (en este

caso, 6) para escalar correctamente. El operador entero de conversión de tipo se utiliza para truncar la parte flotante (la parte que se encuentra después del número decimal) de cada valor producido por la expresión anterior. Después *desplazamos* el rango de números producidos sumando 1 al resultado anterior, como en

```
1 + (int) ( Math.random() * 6 )
```

La figura 25.11 confirma que los resultados se encuentran en el rango de 1 a 6.

```

1 // Figura 25.11: EntAleatorio.java
2 // Enteros aleatorios escalados y desplazados
3 import javax.swing.JOptionPane;
4
5 public class EntAleatorio {
6     public static void main( String args[] )
7     {
8         int valor;
9         String salida = "";
10
11        for ( int i = 1; i <= 20; i++ ) {
12            valor = 1 + (int) ( Math.random() * 6 );
13            salida += valor + " ";
14
15            if ( i % 5 == 0 )
16                salida += "\n";
17        }
18
19        JOptionPane.showMessageDialog( null, salida,
20             "20 números aleatorios del 1 al 6",
21             JOptionPane.INFORMATION_MESSAGE );
22
23        System.exit( 0 );
24    } // fin de main
25 } // fin de la clase EntAleatorio

```

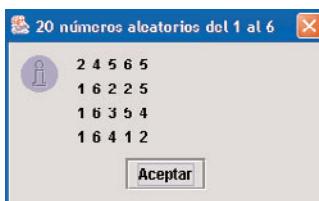


Figura 25.11 Enteros aleatorios escalados y desplazados.

Para mostrar que estos números aparecen con aproximadamente la misma posibilidad, simulemos 6000 tiros de un dado con el programa de la figura 25.12. Cada entero de 1 a 6 debe aparecer aproximadamente 1000 veces.

```

1 // Figura 25.12: TiraDados.java
2 // Tira 6000 veces un dado de seis lados
3 import javax.swing.*;
4
5 public class TiraDados {
6     public static void main( String args[] )
7     {
8         int frecuencial = 0, frecuencia2 = 0,

```

Figura 25.12 Tiro de un dado 6000 veces. (Parte 1 de 2.)

```

9         frecuencia3 = 0, frecuencia4 = 0,
10        frecuencia5 = 0, frecuencia6 = 0, cara;
11
12        // resume los resultados
13        for ( int tiro = 1; tiro <= 6000; tiro++ ) {
14            cara = 1 + (int) ( Math.random() * 6 );
15
16            switch ( cara ) {
17                case 1:
18                    ++frecuencial;
19                    break;
20                case 2:
21                    ++frecuencia2;
22                    break;
23                case 3:
24                    ++frecuencia3;
25                    break;
26                case 4:
27                    ++frecuencia4;
28                    break;
29                case 5:
30                    ++frecuencia5;
31                    break;
32                case 6:
33                    ++frecuencia6;
34                    break;
35            } // fin de switch
36        } // fin de for
37
38        JTextArea areaSalida = new JTextArea( 7, 10 );
39
40        areaSalida.setText(
41            "Cara\tFrecuencia" +
42            "\n1\t" + frecuencial +
43            "\n2\t" + frecuencia2 +
44            "\n3\t" + frecuencia3 +
45            "\n4\t" + frecuencia4 +
46            "\n5\t" + frecuencia5 +
47            "\n6\t" + frecuencia6 );
48
49        JOptionPane.showMessageDialog( null, areaSalida,
50            "Lanzando 6000 veces un dado",
51            JOptionPane.INFORMATION_MESSAGE );
52        System.exit( 0 );
53    } // fin de main
54} // fin de la clase TiraDados

```

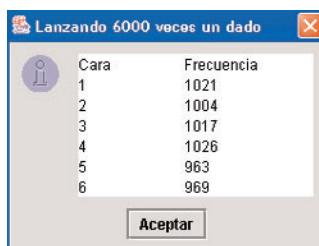


Figura 25.12 Tiro de un dado 6000 veces. (Parte 2 de 2.)

Como muestra la salida del programa, al escalar y al desplazar hemos utilizado el método `random` para simular de manera real el tiro de un dado. El ciclo `for` de la línea 13 itera 6000 veces. Durante cada iteración del ciclo, la línea 14 produce un valor entre 1 y 6. La estructura `switch` anidada de la línea 16 utiliza el valor `cara` que se eligió al azar como su expresión de control. Basándose en el valor de `cara`, una de las seis variables contadores se incrementa durante cada iteración del ciclo. Observe que *no* se proporciona ningún caso `default` en la estructura `switch`. Después de que estudiemos los arreglos en las secciones 25.9 y 25.10, mostraremos cómo remplazar toda la estructura `switch` de este programa con una instrucción de una sola línea. Ejecute el programa varias veces y observe los resultados. Vea que se obtiene una secuencia *diferente* de números aleatorios cada vez que se ejecuta el programa, por lo que los resultados de éste deben variar.

Anteriormente mostramos cómo escribir una sola instrucción para simular el tiro de un dado, por medio de la instrucción

```
cara = 1 + (int) ( Math.random() * 6 );
```

la cual siempre asigna un entero (al azar) a la variable `cara`, en el rango $1 \leq \text{cara} \leq 6$. Observe que el ancho de este rango (es decir, el número de enteros consecutivos en el rango) es 6, y que el número inicial del rango es 1. Considerando la instrucción anterior, vemos que el ancho del rango es determinado por el número utilizado para escalar `random` con el operador de multiplicación (es decir, 6), y que el número inicial del rango es igual que el número (es decir, 1) sumado a `(int) (Math.random() * 6)`. Podemos generalizar este resultado de la siguiente forma:

```
n = a + (int) ( Math.random() * b );
```

donde `a` es el *valor de desplazamiento* (el cual es igual al primer número del rango deseado de enteros consecutivos) y `b` es el *factor de escalamiento* (el cual es igual al ancho del rango deseado de enteros consecutivos).

25.7 Ejemplo: Un juego de azar

Recuerde nuestro ejemplo del juego de “craps” del capítulo 5. Ahora presentamos una nueva versión del simulador de craps como un applet de Java, en la figura 25.13.

```

1 // Figura 25.13: Craps.java
2 // Craps
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class Craps extends JApplet implements ActionListener {
8     // variables constantes para el estado del juego
9     final int GANA = 0, PIERDE = 1, CONTINUAR = 2;
10
11    // otras variables utilizadas en el programa
12    boolean primerTiro = true;           // verdadero si es el primer tiro
13    int sumaDeDados = 0;                 // suma de los dados
14    int miPunto = 0;                    // punto si no se gana/pierde en el primer tiro
15    int estadoDelJuego = CONTINUAR;    // el juego aún no ha terminado
16
17    // componentes de la interfaz gráfica de usuario
18    JLabel etiquetaDado1, etiquetaDado2, etiquetaSuma, etiquetaPunto;
19    JTextField primerDado, segundoDado, suma, punto;
20    JButton tiro;
21
22    // inicializa los componentes de la interfaz gráfica de usuario
23    public void init()

```

Figura 25.13 Programa para simular el juego de craps. (Parte 1 de 4.)

```
24  {
25      Container c = getContentPane();
26      c.setLayout( new FlowLayout() );
27
28      etiquetaDado1 = new JLabel( "Dado 1" );
29      c.add( etiquetaDado1 );
30      primerDado = new JTextField( 10 );
31      primerDado.setEditable( false );
32      c.add( primerDado );
33
34      etiquetaDado2 = new JLabel( "Dado 2" );
35      c.add( etiquetaDado2 );
36      segundoDado = new JTextField( 10 );
37      segundoDado.setEditable( false );
38      c.add( segundoDado );
39
40      etiquetaSuma = new JLabel( "La suma es" );
41      c.add( etiquetaSuma );
42      suma = new JTextField( 10 );
43      suma.setEditable( false );
44      c.add( suma );
45
46      etiquetaPunto = new JLabel( "El puntaje es" );
47      c.add( etiquetaPunto );
48      punto = new JTextField( 10 );
49      punto.setEditable( false );
50      c.add( punto );
51
52      tiro = new JButton( "Tirar dados" );
53      tiro.addActionListener( this );
54      c.add( tiro );
55  } // fin del método init
56
57  // llama al método jugar, cuando se oprime el botón
58  public void actionPerformed( ActionEvent e )
59  {
60      jugar();
61  } // fin del método actionPerformed
62
63  // procesa un tiro de los dados
64  public void jugar()
65  {
66      if ( primerTiro ) {           // primer tiro de los dados
67          sumaDeDados = tiroDados();
68
69          switch ( sumaDeDados ) {
70              case 7: case 11:           // gana en el primer tiro
71                  estadoDelJuego = GANA;
72                  punto.setText( "" ); // limpia el campo de texto de puntaje
73                  break;
74              case 2: case 3: case 12: // pierde en el primer tiro
75                  estadoDelJuego = PIERDE;
76                  punto.setText( "" ); // limpia el campo de texto de puntaje
77                  break;
78              default:                 // recuerda el puntaje
```

Figura 25.13 Programa para simular el juego de craps. (Parte 2 de 4.)

```

79         estadoDelJuego = CONTINUAR;
80         miPunto = sumaDeDados;
81         punto.setText( Integer.toString( miPunto ) );
82         primerTiro = false;
83         break;
84     } // fin de switch
85 } // fin de if
86 else {
87     sumaDeDados = tiroDados();
88
89     if ( sumaDeDados == miPunto )      // gana por puntos
90         estadoDelJuego = GANA;
91     else
92         if ( sumaDeDados == 7 )          // pierde por tirar 7
93             estadoDelJuego = PIERDE;
94 } // fin de else
95
96 if ( estadoDelJuego == CONTINUAR )
97     showStatus( "Tire de nuevo." );
98 else {
99     if ( estadoDelJuego == GANA )
100        showStatus( "El jugador gana. " +
101                  "Haga clic en Tirar dados para jugar de nuevo." );
102    else
103        showStatus( "El jugador pierde. " +
104                  "Haga clic en Tirar dados para jugar de nuevo." );
105
106    primerTiro = true;
107 } // fin de else
108 // fin del método jugar
109
110 // tirar dados
111 public int tiroDados()
112 {
113     int dado1, dado2, trabajaSuma;
114
115     dado1 = 1 + ( int ) ( Math.random() * 6 );
116     dado2 = 1 + ( int ) ( Math.random() * 6 );
117     trabajaSuma = dado1 + dado2;
118
119     primerDado.setText( Integer.toString( dado1 ) );
120     segundoDado.setText( Integer.toString( dado2 ) );
121     suma.setText( Integer.toString( trabajaSuma ) );
122
123     return trabajaSuma;
124 } // fin del método tiroDados
125 } // fin de la clase Craps

```



Figura 25.13 Programa para simular el juego de craps. (Parte 3 de 4.)

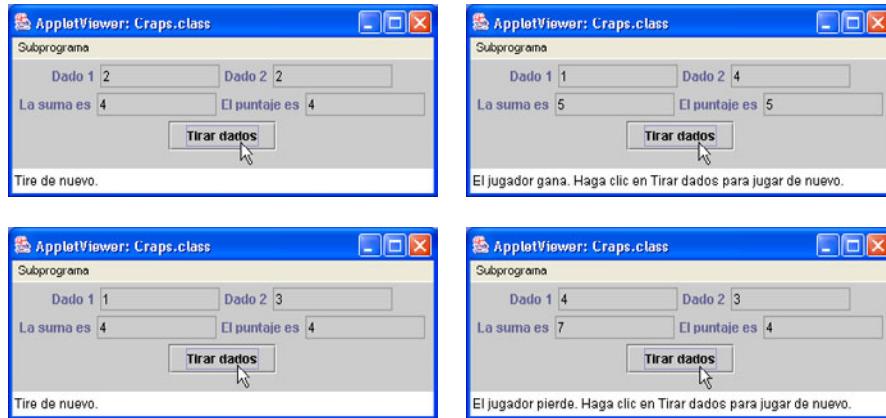


Figura 25.13 Programa para simular el juego de craps. (Parte 4 de 4.)

Observe que así como en la versión de C del simulador, el jugador debe tirar dos dados en el primer tiro y en los subsiguientes. Cuando ejecute el applet, haga clic en el botón **Tirar dados**, para jugar. La esquina inferior izquierda de la ventana del **appletviewer** despliega los resultados de cada tiro. Las capturas de pantalla muestran cuatro ejecuciones separadas del applet (una en donde se gana y otra en donde se pierde en el primer tiro, y una en donde se gana y otra donde se pierde después del primer tiro).

Hasta el momento, todas las interacciones del usuario con aplicaciones y applets han sido a través de un diálogo de entrada (en el que el usuario podía escribir un valor de entrada para el programa), o a través de un diálogo de mensaje (en el que se desplegaba un mensaje para el usuario, y éste podía hacer clic en **Aceptar** para desechar el diálogo). Aunque éstas son formas válidas para recibir la entrada de un usuario y para desplegar resultados en un programa en Java, sus capacidades son bastante limitadas; un diálogo de entrada puede obtener sólo un valor a la vez por parte del usuario, y un diálogo de mensaje puede desplegar sólo un mensaje. Es mucho más común recibir múltiples entradas a la vez por parte del usuario (como la información sobre el nombre y la dirección del usuario), o desplegar muchas piezas de datos a la vez (en este ejemplo, los valores de los dados, la suma y el puntaje). Para comenzar nuestra introducción sobre interfaces de usuario más elaboradas, este programa ilustra dos nuevos conceptos sobre la interfaz gráfica de usuario; cómo adjuntar diversos componentes GUI a un applet, y la *manipulación de eventos* de la interfaz gráfica de usuario.

Las líneas 3 a 5

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

especifican al compilador en dónde localizar a las clases utilizadas en este applet. La primera **import** especifica que el programa utiliza clases del paquete **java.awt** (específicamente las clases **Container** y **FlowLayout**). La segunda **import** especifica que el programa utiliza clases del paquete **java.awt.event**. Este paquete contiene muchos tipos de datos que permiten a un programa procesar las interacciones de un usuario con la GUI de un programa. En este programa, utilizamos los tipos de datos **ActionListener** y **ActionEvent** del paquete **java.awt.event**. La última instrucción **import** especifica que el programa utiliza clases del paquete **javax.swing** (específicamente las clases **JApplet**, **JLabel**, **JTextField** y **JButton**).

Como dijimos antes, todo programa en Java se basa en al menos una definición de clase que amplía y mejora la definición de una clase existente, a través de la herencia. Recuerde que las applets se heredan de la clase **JApplet**. La línea 7

```
public class Craps extends JApplet implements ActionListener {
```

indica que la clase **Craps** se hereda de **JApplet** e implementa (*implements*) **ActionListener**. Una clase puede heredar atributos y comportamientos (datos y métodos) de otra clase especificada a la derecha de la palabra reservada **extends**, en la definición de la clase. Además, una clase puede implementar una o más *in-*

interfaces. Una interfaz especifica uno o más comportamientos (es decir, métodos) *que usted debe definir* en la definición de la clase. La interfaz **ActionListener** especifica que esta clase *debe definir* un método con la primera línea

```
public void actionPerformed( ActionEvent e )
```

En este ejemplo, esta tarea del método es para procesar una interacción del usuario con el **JButton** (llamado **Tirar dados** en la interfaz de usuario). Cuando el usuario oprime el botón, éste método es invocado automáticamente, en respuesta a la interacción del usuario. A este proceso se le conoce como *manipulación de eventos*. El *evento* es la interacción del usuario (quien oprime el botón). El *manipulador de eventos* es el método **actionPerformed**, al cual se invoca automáticamente en respuesta al evento. Un poco más adelante explicaremos los detalles de esta interacción y del método **actionPerformed**. El capítulo 27 explica con detalle las interfaces. Por ahora, imite las características que ilustramos que soportan la manipulación de eventos de los componentes GUI que presentamos.

Este juego está razonablemente involucrado. El jugador puede ganar o perder en el primer tiro, o puede ganar o perder en cualquier tiro. La línea 9 del programa

```
final int GANA = 0, PIERDE = 1, CONTINUAR = 2;
```

crea variables que definen los tres estados de un juego de craps; juego ganado, juego perdido o continuar el tiro de dados. La palabra reservada **final**, al principio de la declaración, indica que éstas son *variables constantes*. Las variables constantes deben inicializarse una vez, antes de que se utilicen, y no pueden modificarse después. Con frecuencia, a las variables constantes se les conoce como *constantes nombradas* o *variables de sólo lectura*.

Error común de programación 25.12



Después de que se inicializó una variable **final**, intentar asignar otro valor a esa variable es un error de sintaxis.



Buena práctica de programación 25.5

Sólo utilice letras mayúsculas (con guiones bajos entre las palabras) en los nombres de variables **final**. Esto hace que las constantes resalten en un programa.



Buena práctica de programación 25.6

Utilizar variables **final** con nombres descriptivos, en lugar de utilizar constantes enteras (como 2), hace que los programas sean legibles.

Las líneas 12 a 15

```
boolean primerTiro = true; // verdadero si es el primer tiro
int sumaDeDados = 0; // suma de los dados
int miPunto = 0; // punto si no se gana/pierde en el primer tiro
int estadoDelJuego = CONTINUAR; // el juego aún no ha terminado
```

declaran diversas variables de instancia que se utilizan a lo largo del applet **Craps**. La variable **primerTiro** indica si el siguiente tiro de los dados es el primero del juego actual. La variable **sumaDeDados** mantiene la suma de los dados correspondiente al último tiro. La variable **miPunto** almacena el “punto”, si el jugador no gana o pierde en el primer tiro. La variable **estadoDelJuego** da seguimiento al estado actual del juego (**GANA**, **PIERDE** o **CONTINUAR**).

Las líneas 18 a 20

```
JLabel etiquetaDado1, etiquetaDado2, etiquetaSuma, etiquetaPunto;
JTextField primerDado, segundoDado, suma, punto;
JButton tiro;
```

declara referencias hacia los componentes GUI utilizados en la interfaz gráfica de usuario de este applet. Las referencias **etiquetaDado1**, **etiquetaDado2**, **etiquetaSuma** y **etiquetaPunto** se refieren a objetos **JLabel**. Una **JLabel** contiene una cadena de caracteres a desplegar en la pantalla. Por lo general, una **JLabel** indica el propósito de otro elemento de la interfaz gráfica de usuario en la pantalla. En las capturas de pantalla de la figura 25.13, los objetos **JLabel** son el texto que se encuentra a la izquierda de cada rectángulo en las primeras dos filas de la interfaz de usuario. Las referencias **primerDado**, **segundoDado**, **su-**

ma y **punto** se refieren a objetos **JTextField**. Los **JTextField** se utilizan para obtener una sola línea de información desde el teclado por parte del usuario, o para desplegar información en la pantalla. En las capturas de pantalla de la figura 25.13, los objetos **JTextField** son los rectángulos que se encuentran a la derecha de cada **JLabel**, en las dos primeras filas de la interfaz de usuario. La referencia **tiro** se refiere a un objeto **JButton**. Cuando el usuario oprime un **JButton**, por lo general el programa responde realizando una tarea (en este ejemplo, tirando los dados). El objeto **JButton** es el rectángulo que contiene las palabras **Tira dados**, en la parte inferior de la interfaz de usuario de la figura 25.13. En ejemplos anteriores ya utilizamos **JtextFields** y **JButtons**. Todo mensaje de diálogo y todo diálogo de entrada contenía un botón **Aceptar** para desechar el diálogo de mensaje, o para enviar la entrada del usuario al programa. Todo diálogo de entrada también contenía un **JTextField**, en el que el usuario escribía un valor de entrada.

El método **init** (línea 23) crea los objetos componentes GUI y los adjunta a la interfaz de usuario. La línea 25

```
Container c = getContentPane();
```

declara una referencia **c** de **Container**, y le asigna el resultado de una llamada al método **getContentPane**. Recuerde, el método **getContentPane** devuelve una referencia al panel de contenido del applet que puede utilizarse para adjuntar componentes GUI a la interfaz de usuario del applet.

La línea 26

```
c.setLayout( new FlowLayout() );
```

utiliza el método **setLayout** de **Container** para definir un *administrador de diseño* para la interfaz de usuario del applet. Los administradores de diseño se proporcionan para acomodar los componentes GUI en un **Container**, para efectos de presentación. Estos administradores determinan la posición y el tamaño de cada componente GUI adjunto al contenedor. Esto permite al programador concentrarse en la “apariencia visual” básica, y deja a los administradores de diseño el procesamiento de la mayoría de los detalles del diseño.

FlowLayout es el administrador de diseño más básico. Los componentes GUI se colocan en un **Container** de izquierda a derecha, en el orden en el que se adjuntan al **Container** por medio del método **add**. Cuando se alcanza el borde del contenedor, los componentes continúan en la siguiente línea. La instrucción anterior crea un nuevo objeto de la clase **FlowLayout**, y lo pasa al método **setLayout**. En general, el diseño se establece antes de que cualquier componente GUI se agregue al **Container**.

[*Nota:* Cada **Container** puede tener sólo un administrador de diseño a la vez (**Containers** separados en el mismo programa pueden tener diferentes administradores de diseño). La mayoría de los ambientes de programación en Java proporcionan herramientas de diseño GUI que ayudan al programador a diseñar de manera gráfica una GUI, y después automáticamente se escribe código Java para crear la GUI. Algunos de estos diseñadores GUI también permiten al programador utilizar los administradores de diseño. El capítulo 29 explica diversos administradores de diseño, que permiten un control más preciso sobre el diseño de los componentes GUI.]

Las líneas 28 a 32, 34 a 38, 40 a 44, y 46 a 50 crean un par de **JLabel** y **JTextField**, y lo adjuntan a la interfaz de usuario. Estas líneas son muy parecidas, por lo que no concentraremos en las líneas 28 a 32.

```
etiquetaDado1 = new JLabel( "Dado 1" );
c.add( etiquetaDado1 );
primerDado = new JTextField( 10 );
primerDado.setEditable( false );
c.add( primerDado );
```

La línea 28 crea un nuevo objeto **JLabel**, lo inicializa con la cadena **“Dado 1”**, y asigna el objeto a la referencia **etiquetaDado1**. Esto etiqueta al **JTextField** **primerDado** correspondiente en la interfaz de usuario, por lo que el usuario puede determinar el propósito del valor desplegado en **primerDado**. La línea 29 adjunta la **JLabel** a la que **etiquetaDado1** hace referencia en el panel de contenido del applet. La línea 30 crea un nuevo objeto **JTextField**, lo inicializa para que sea de 10 caracteres de ancho, y asigna el objeto a la referencia **primerDado**. Este **JTextField** desplegará el valor del primer dado después de cada tiro de dados. La línea 31 utiliza el método **setEditable** de **JTextField** con el argumento **false** para indicar que el usuario no debe poder escribir en el **JTextField** (es decir, hace que el **JTextField** sea *ineditable*). Un **JTextField** no editable tiene de manera predeterminada un fondo gris (como se aprecia en

los diálogos de entrada). La línea 32 adjunta el **JTextField** a donde hace referencia **primerDado** en el panel de contenido del applet.

La línea 52

```
tiro = new JButton( "Tirar dados" );
```

crea un nuevo objeto **JButton**, lo inicializa con la cadena "**Tira dados**" (esta cadena aparecerá en la parte inferior), y asigna el objeto a la referencia **tiro**.

La línea 53

```
tiro.addActionListener( this );
```

especifica que *este (this)* applet debe *escuchar* los eventos de **tiro** de **JButton**. La palabra reservada **this** permite al applet hacer referencia a sí mismo (en el capítulo 26 explicaremos con detalle a **this**). Cuando el usuario interactúa con un componente GUI, se envía un *evento* al applet. Los eventos GUI son mensajes que indican que el usuario del programa interactuó con uno de los componentes GUI del programa. Por ejemplo, cuando en este programa oprime el **JButton tiro**, se envía un evento al applet que indica que el usuario oprimió el botón. Esto le indica al applet que el *usuario realizó una acción* en el **JButton**, y automáticamente llama al método **actionPerformed** para que procese la interacción del usuario.

A este estilo de programación se le conoce como *programación manejada por eventos*; el usuario interactúa con un componente GUI, al programa se le notifica el evento y lo procesa. La interacción del usuario con la GUI "maneja" el programa. Los métodos que son llamados cuando ocurre un evento también son conocidos como *métodos para manejo de eventos*. Cuando ocurre un evento GUI en un programa, Java crea un objeto que contenga la información sobre el evento que ocurrió, y *automáticamente llama* a un método para manejo de eventos apropiado. Antes de que pueda procesarse cualquier evento, cada componente GUI debe saber cuál objeto del programa define el método para manejo de eventos que se llamará cuando ocurra un evento. En la línea 53, se utiliza el método **addActionListener** de **JButton** para decirle a **tiro** que el applet (**this**) puede escuchar *eventos de acción*, y define un método **actionPerformed**. A esto se le conoce como *registro del manipulador de eventos* con el componente GUI (también quisiéramos llamarlo la línea que *empieza a escuchar*, ya que el applet ahora está escuchando los eventos del botón). Para responder a un evento de acción, debemos definir una clase que implemente un **ActionListener** (esto requiere que la clase también defina un método **actionPerformed**) y debemos registrar el manipulador de eventos con el componente GUI. Por último, la última línea de **init** adjunta el **JButton** al que **tiro** hace referencia en el panel de contenido del applet, con lo que se completa la interfaz de usuario.

El método **actionPerformed** (línea 58) es uno de diversos métodos que procesan las interacciones entre el usuario y los componentes GUI. La primera línea del método

```
public void actionPerformed( ActionEvent e )
```

indica que **actionPerformed** es un método **public** que devuelve nada (**void**) cuando completa su tarea. Cuando se llama automáticamente, el método **actionPerformed** recibe un argumento (un **ActionEvent**), en respuesta a una acción realizada por el usuario sobre un componente GUI (en este caso, oprimir el **JButton**). El argumento **ActionEvent** contiene información acerca de la acción que ocurrió.

Definimos un método **tiraDados** (línea 111) para tirar los dados y para calcular y desplegar su suma. El método **tiraDados** se define una vez, pero se le llama desde dos lugares del programa (líneas 67 y 87). El método **tiraDados** no toma argumentos, por lo que tiene una lista de parámetros vacía. El método **tiraDados** devuelve la suma de los dos dados, por lo que en el encabezado del método se indica un tipo de retorno **int**.

El usuario hace clic en el botón "**Tira Dados**" para realizar su tiro. Esto invoca al método **actionPerformed** (línea 58) del applet, el cual después invoca al método **jugar** (definido en la línea 64). El método **jugar** verifica la variable **booleana primerTiro** (línea 66) para determinar si es **true** o **false**. Si es **true**, éste es el primer tiro del juego. La línea 67 llama a **tiraDados** (definido en la línea 111), el cual escoge dos valores al azar entre 1 y 6, despliega el valor del primer dado, el del segundo dado y la suma de los dos en los tres primeros **JTextFields**, y devuelve la suma de los dados. Observe que los valores enteros se convierten en **Strings** con el método **static Integer.toString**, ya que los **JTextFields** sólo

pueden desplegar cadenas. Después del primer tiro, la estructura **switch** anidada de la línea 69 determina si se ganó o se perdió el juego, o si el juego debe continuar con otro tiro. Después del primer tiro, si el juego no terminó, la suma se guarda en **miPunto** y se despliega en el **JTextField punto**.

El programa continúa con la estructura **if/else** anidada de la línea 96, la cual utiliza el método **showStatus** para desplegar en la barra de estado del **appletviewer**

Tira de nuevo.

si el **estadoDelJuego** es igual que **CONTINUAR** y

El jugador gana. Haga clic en Tira Dados para jugar otra vez.

si el **estadoDelJuego** es igual que **GANAR** y

El jugador pierde. Haga clic en Tira Dados para jugar otra vez.

si el **estadoDelJuego** es igual que **PIERDE**. El método **showStatus** recibe un argumento **String** y lo despliega en la barra de estado del **appletviewer** o navegador. Si se ganó o se perdió el juego, la línea 106 establece en **true** a **primerTiro**, para indicar que el siguiente tiro de dados es el primero del siguiente juego.

El programa entonces espera que el usuario nuevamente haga clic en el botón “**Tira Dados**”. Cada vez que el usuario presione este botón, el método **actionPerformed** invoca al método **jugar** y el método **tiraDados** es llamado para producir una nueva **suma**. Si la **suma** coincide con **miPunto**, el **estadoDelJuego** se establece en **GANAR**, la estructura **if/else** de la línea 96 se ejecuta y el juego se completa. Si la **suma** es igual que **7**, **estadoDelJuego** se establece en **PIERDE**, la estructura **if/else** de la línea 96 se ejecuta y el juego se completa. Al hacer clic en el botón “**Tira Dados**” se inicia un nuevo juego. A lo largo del programa, los cuatro **JTextFields** se actualizan con los nuevos valores de los dados y la suma en cada tiro, y el **JTextField** punto se actualiza cada vez que inicia un nuevo juego.

25.8 Métodos de la clase **JApplet**

Hasta este punto del texto hemos escrito muchos applets, pero aún no hemos explicado los métodos clave de la clase **JApplet** que son llamados automáticamente durante la ejecución de un applet. La figura 25.14 lista los métodos clave de la clase **JApplet**, cuándo se les llama, y el propósito de cada uno.

Estos métodos de **JApplet** son definidos por la API de Java para que no hagan cosa alguna, a menos que usted proporcione una definición en la definición de la clase de su applet. Si quisiera utilizar uno de estos métodos en un applet que está definiendo, debe definir la primera línea del método como muestra la figura 25.14. De lo contrario, el método no será llamado automáticamente durante la ejecución del applet.

Método	Cuándo se llama al método y su propósito
public void init()	A este método lo llama una vez el appletviewer o el navegador cuando se carga un applet para su ejecución. Éste realiza la inicialización de un applet. Las acciones típicas que se realizan aquí son la inicialización de variables de instancia y de componentes GUI del applet, la carga de sonidos a reproducir o imágenes a desplegar (capítulo 30), y la creación de subprocessos.
public void start()	Este método es llamado después de que el método init completa su ejecución, y cada vez que el usuario del navegador regresa a la página HTML en la que reside el applet (después de explorar otra página HTML). Este método realiza cualquier tarea que deba completarse cuando el applet se carga por primera vez en el navegador, y que deba realizarse cada vez que la página HTML en la que reside el applet se vuelva a visitar. Las acciones típicas que se realizan aquí incluyen iniciar una animación (capítulo 30), e iniciar otros subprocessos de ejecución.

Figura 25.14 Métodos de **JApplet** que se llaman automáticamente durante la ejecución de un applet. (Parte 1 de 2.)

Método	Cuándo se llama al método y su propósito
<code>public void paint(Graphics g)</code>	Este método es llamado para dibujar en el applet, después de que el método <code>init</code> completa su ejecución y después de que el método <code>start</code> ha iniciado su ejecución. También se le llama automáticamente cada vez que el applet necesita repintarse. Por ejemplo, si el usuario del navegador cubre el applet con otra ventana abierta en la pantalla, entonces descubre el applet, y se llama al método <code>paint</code> . Las acciones típicas realizadas aquí involucran el dibujo con el objeto <code>g</code> de <code>Graphics</code> , el cual es pasado al método <code>paint</code> .
<code>public void stop()</code>	Este método es llamado cuando el applet debe detener su ejecución; normalmente cuando el usuario del navegador abandona la página HTML en la que el applet reside. Este método realiza cualquier tarea necesaria para suspender la ejecución del applet. Las acciones típicas realizadas aquí son detener la ejecución de animaciones y subprocessos.
<code>public void destroy()</code>	Este método es llamado cuando el applet está siendo removido de la memoria; normalmente cuando el usuario del navegador abandona la sesión de navegación. Este método realiza cualquier tarea necesaria para destruir recursos asignados al applet.

Figura 25.14 Métodos de `JApplet` que se llaman automáticamente durante la ejecución de un applet. (Parte 2 de 2.)



Error común de programación 25.13

Proporcionar una definición para uno de los métodos de `JApplet init, start, paint, stop o destroy`, que no coincide con los encabezados de los métodos que muestra la figura 25.14, dará como resultado un método que no será llamado automáticamente durante la ejecución del applet.

El método `repaint` también es de interés para muchos programadores de applets. El método `paint` del applet por lo general se invoca automáticamente. ¿Qué sucedería si quisiera cambiar la apariencia del applet, en respuesta a interacciones del usuario con el applet? En tales situaciones, podría desear llamar directamente a `paint`. Sin embargo, para llamar a `paint`, debemos pasarle el parámetro `Graphics` que espera. Esto implica un problema para nosotros. Nosotros no tenemos un objeto `Graphics` a nuestra disposición para pasarlo a `paint` (en el capítulo 30 explicaremos este asunto). Por esta razón, se le proporciona el método `repaint`. La instrucción

```
repaint();
```

invoca otro método llamado `update`, y le pasa el objeto `Graphics` por usted. El método `update` borra cualquier dibujo que se hubiera hecho anteriormente en el applet, después invoca al método `paint` y le pasa el objeto `Graphics` por usted. En el capítulo 30 explicamos con detalle los métodos `repaint` y `update`.

25.9 Declaración y asignación de arreglos

Los arreglos ocupan espacio en memoria. El programador especifica el tipo de los elementos y utiliza el operador `new` para asignar dinámicamente el número de elementos requeridos por cada arreglo. Los arreglos se asignan con `new`, debido a que éstos se consideran como objetos, y todos los objetos deben crearse con `new`. Para asignar 12 elementos al arreglo entero `c`, se utiliza la declaración

```
int c[] = new int[12];
```

La instrucción anterior también puede realizarse en dos pasos, de la siguiente manera:

```
int c[];           // declara el arreglo
c = new int[ 12 ]; // asigna el arreglo
```

Cuando los arreglos se asignan, los elementos se inicializan en cero, si se trata de variables de tipos de datos primitivos, en **false**, si se trata de variables **boolean**, o en **null**, si se trata de referencias (de cualquier tipo que no sea primitivo).

Error común de programación 25.14



A diferencia de C o de C++, el número de elementos en el arreglo nunca se especifica entre corchetes después del nombre del arreglo en una declaración. La declaración `int c[12];` ocasiona un error de sintaxis.

La memoria puede reservarse para diversos arreglos con una sola declaración. La siguiente declaración reserva 100 elementos para el arreglo **String b**, y 27 elementos para el arreglo **String x**:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

Cuando se declara un arreglo, al principio de la declaración podemos combinar el tipo del arreglo y los corchetes, para indicar que todos los identificadores en la declaración representan arreglos, como en

```
double[] arreglo1, arreglo2;
```

la cual declara tanto el **arreglo1** como el **arreglo2** como valores **double**. Como mostramos anteriormente, la declaración e inicialización de un arreglo pueden combinarse en la declaración. La siguiente declaración reserva 10 elementos para el **arreglo1** y 20 elementos para el **arreglo2**:

```
double[] arreglo1 = new double[ 10 ], arreglo2 = new double[ 20 ];
```

Los arreglos pueden declararse para que contengan cualquier tipo de dato. Es importante recordar que en un arreglo de tipo de datos primitivos, cada elemento contiene un valor del tipo de datos declarado para el arreglo. Sin embargo, en un arreglo de tipo no primitivo, cada elemento es una referencia hacia un objeto del tipo del arreglo. Por ejemplo, cada elemento de un arreglo **String** es una referencia hacia una **String** que tiene de manera predeterminada el valor **null**.

25.10 Ejemplos del uso de arreglos

La aplicación de la figura 25.15 utiliza el operador **new** para asignar dinámicamente un arreglo de 10 elementos que se inicializan en cero, y después imprime el arreglo en formato tabular.

```

1 // Figura 25.15: InicArreglo.java
2 // inicializa un arreglo
3 import javax.swing.*;
4
5 public class InicArreglo {
6     public static void main( String args[] )
7     {
8         String salida = "";
9         int n[];           // declara una referencia a un arreglo
10
11         n = new int[ 10 ]; // asigna dinámicamente el arreglo
12
13         salida += "Subíndice\tValor\n";
14
15         for ( int i = 0; i < n.length; i++ )
16             salida += i + "\t" + n[ i ] + "\n";
17
18         JTextArea areaSalida = new JTextArea( 11, 10 );
19         areaSalida.setText( salida );
20
21         JOptionPane.showMessageDialog( null, areaSalida,

```

Figura 25.15 Inicialización en cero de los elementos de un arreglo. (Parte 1 de 2.)

```

22     "Inicializa un arreglo con valores int",
23     JOptionPane.INFORMATION_MESSAGE );
24
25     System.exit( 0 );
26 } // fin de main
27 } // fin de la clase InicArreglo

```

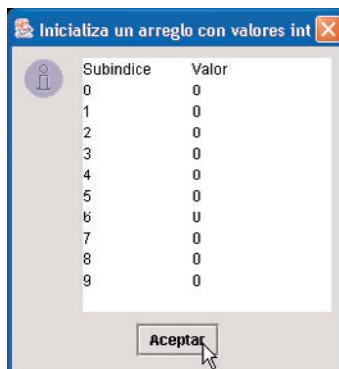


Figura 25.15 Inicialización en cero de los elementos de un arreglo. (Parte 2 de 2.)

La línea 9 declara a **n** como una referencia capaz de referirse a un arreglo de enteros. La línea 11 asigna los 10 elementos del arreglo con **new**, e inicializa la referencia. La línea 13 agrega a **String salida** los encabezados para las columnas de la salida desplegada por el programa.

Las líneas 15 y 16

```

for ( int i = 0; i < n.length; i++ )
    salida += i + "\t" + n[ i ] + "\n";

```

utilizan una estructura **for** para construir la **String** de **salida** que se desplegará en un **JTextArea** de un diálogo de mensaje. Observe el uso de la cuenta basada en cero (recuerde, los subíndices parten de 0), por lo que el ciclo puede acceder a cada elemento del arreglo. Además, observe la expresión **n.length** en la condición de la estructura **for** para determinar la longitud del arreglo. En este ejemplo, la longitud del arreglo es 10, por lo que el ciclo continúa en ejecución mientras el valor de la variable de control **i** sea menor que 10. Para un arreglo de 10 elementos, los valores de los subíndices van de 0 a 9, por lo que utilizar el operador de menor que (**<**) garantiza que el ciclo no intentará acceder a un elemento que se encuentre más allá del final del arreglo. Los elementos de un arreglo pueden asignarse e inicializarse en la declaración del arreglo, colocando después de la declaración un signo de igual y una *lista de inicializadores* separada por comas entre llaves (**{** y **}**). En este caso, el tamaño del arreglo se determina por medio del número de elementos en la lista de inicialización. Por ejemplo, la instrucción

```
int n[] = { 10, 20, 30, 40, 50 };
```

crea un arreglo de cinco elementos con los subíndices **0, 1, 2, 3** y **4**. Observe que la declaración anterior no requiere el operador **new** para crear el objeto arreglo; esto lo proporciona el compilador siempre que encuentra una declaración de arreglo que incluye una lista de inicialización.

La aplicación de la figura 25.16 inicializa un arreglo entero con 10 valores (línea 12) y lo despliega en formato tabular en un **JTextArea** de un diálogo de mensaje.

```

1 // Figura 25.16: InitArray.java
2 // inicializa un arreglo mediante una declaración
3 import javax.swing.*;
4

```

Figura 25.16 Inicialización de los elementos de un arreglo por medio de una declaración. (Parte 1 de 2.)

```

5  public class InicArreglo {
6      public static void main( String args[] )
7      {
8          String salida = "";
9
10         // La lista de inicialización especifica el número de elementos y
11         // el valor de cada elemento.
12         int n[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
13
14         salida += "Subindice\tValor\n";
15
16         for ( int i = 0; i < n.length; i++ )
17             salida += i + "\t" + n[ i ] + "\n";
18
19         JTextArea areaSalida = new JTextArea( 11, 10 );
20         areaSalida.setText( salida );
21
22         JOptionPane.showMessageDialog( null, areaSalida,
23             "Inicializa un arreglo mediante una declaracion",
24             JOptionPane.INFORMATION_MESSAGE );
25
26         System.exit( 0 );
27     } // fin de main
28 } // fin de la clase InicArreglo

```

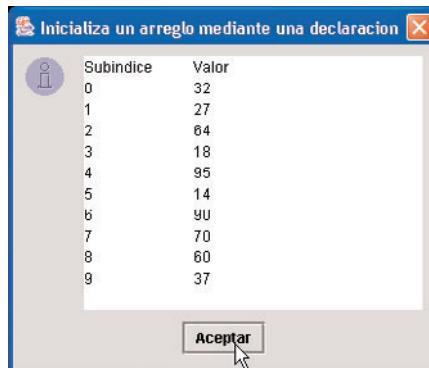


Figura 25.16 Inicialización de los elementos de un arreglo por medio de una declaración. (Parte 2 de 2.)

La aplicación de la figura 25.17 inicializa los elementos del arreglo **n** de 10 elementos con los enteros pares **2, 4, 6, ..., 20** y lo imprime en formato tabular. Estos números se generan multiplicando cada valor sucesivo del contador del ciclo por **2** y sumándole **2**.

```

1 // Figura 25.17: InicArreglo.java
2 // inicializa el arreglo n con los enteros pares de 2 a 20
3 import javax.swing.*;
4
5 public class InicArreglo {
6     public static void main( String args[] )
7     {
8         final int TAMANIO_ARREGLO = 10;
9         int n[]; // referencia a un arreglo de enteros

```

Figura 25.17 Generación de valores para colocarlos como elementos de un arreglo. (Parte 1 de 2.)

```

10     String salida = "";
11
12     n = new int[ TAMANIO_ARREGLO ]; // asigna el arreglo
13
14     // Establece los valores en el arreglo
15     for ( int i = 0; i < n.length; i++ )
16         n[ i ] = 2 + 2 * i;
17
18     salida += "Subindice\tValor\n";
19
20     for ( int i = 0; i < n.length; i++ )
21         salida += i + "\t" + n[ i ] + "\n";
22
23     JTextArea areaSalida = new JTextArea( 11, 10 );
24     areaSalida.setText( salida );
25
26     JOptionPane.showMessageDialog( null, areaSalida,
27         "Inicializa a numeros pares de 2 a 20",
28         JOptionPane.INFORMATION_MESSAGE );
29
30     System.exit( 0 );
31 } // fin de main
32 } // fin de la clase InicArreglo

```

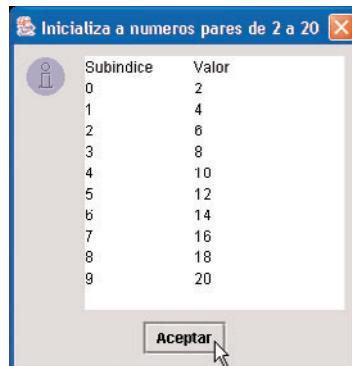


Figura 25.17 Generación de valores para colocarlos como elementos de un arreglo. (Parte 2 de 2.)

La línea 8

```
final int TAMANIO ARREGLO = 10;
```

utiliza el calificador **final** para declarar una variable constante llamada **TAMANIO_ARREGLO**, cuyo valor es **10**. Las variables constantes deben inicializarse antes de utilizarlas, y no pueden modificarse después. Si se intenta modificar una variable **final** después de declararla como muestra la instrucción anterior, el compilador despliega un mensaje como

Can't assign a value to a final variable

Si se intenta modificar una variable **final** después de que se declara, y después se inicializa en una instrucción separada, el compilador despliega un mensaje de error como

Can't assign a second value to a blank final variable

Si se intenta utilizar una variable local **final** antes de que se inicialice, el compilador despliega el mensaje de error

Variable `nombreVariable` may not have been initialized

```

1 // Figura 25.18: SumaArreglo.java
2 // Cálculo de la suma de los elementos de un arreglo
3 import javax.swing.*;
4
5 public class SumaArreglo {
6     public static void main( String args[] )
7     {
8         int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9         int total = 0;
10
11        for ( int i = 0; i < a.length; i++ )
12            total += a[ i ];
13
14        JOptionPane.showMessageDialog( null,
15             "Total de elementos en el arreglo: " + total,
16             "Suma de los elementos del arreglo",
17             JOptionPane.INFORMATION_MESSAGE );
18
19        System.exit( 0 );
20    } // fin de main
21 } // fin de la clase SumaArreglo

```



Figura 25.18 Cálculo de la suma de los elementos de un arreglo.

Si se intenta utilizar una variable de instancia **final** antes de inicializarla, el compilador despliega el mensaje de error

```
Blank final variable 'nombreVariable' may not have been initialized. It must be assigned a value in an initializer, or in every constructor.
```

Las variables constantes también son conocidas como *constantes nombradas* o *variables de sólo lectura*. Con frecuencia se utilizan para hacer que un programa sea más legible. Observe que el término “constante variable” es un oxímoron (una contradicción en términos) como “chico grande” o “bien mal”.

Error común de programación 25.15



Asignar un valor a una constante variable después de inicializarla, es un error de sintaxis.

La aplicación de la figura 25.18 suma los valores contenidos en el arreglo entero **a** de 10 elementos (declarado, asignado e inicializado en la línea 8). La instrucción (línea 12) en el cuerpo del ciclo **for** hace la totalización. Es importante recordar que los valores que se proporcionaron como inicializadores para el arreglo **a** normalmente se leerían en el programa. Por ejemplo, en un applet, el usuario podría introducir los valores a través de un **JTextField**, o en una aplicación los valores podrían leerse desde un archivo en disco.

Nuestro siguiente ejemplo utiliza arreglos para resumir los resultados de los datos obtenidos en una encuesta. Considere el problema:

Se les pidió a cuarenta estudiantes que calificaran la calidad de la comida de una cafetería para estudiantes en una escala de 1 a 10 (1 significa terrible, y 10 significa excelente). Coloque las 40 respuestas en un arreglo entero y totalice los resultados de la encuesta.

Ésta es una aplicación típica del procesamiento de arreglos (vea la figura 25.19). Querríamos resumir el número de respuestas de cada tipo (es decir, 1 a 10). El arreglo **respuestas** es un arreglo entero de 40 elementos que contiene las respuestas de los estudiantes a la encuesta. Utilizamos un arreglo **frecuencia** de 11 elementos

```

1 // Figura 25.19: EncEstudiantes.java
2 // Programa de encuesta a estudiantes
3 import javax.swing.*;
4
5 public class EncEstudiantes {
6     public static void main( String args[] )
7     {
8         int respuestas[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
9                             1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
10                            6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
11                            5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
12     int frecuencia[] = new int[ 11 ];
13     String salida = "";
14
15     for ( int contestacion = 0;                      // inicializa
16           contestacion < respuestas.length;          // condición
17           contestacion++ )                         // incremento
18           ++frecuencia[ respuestas[ contestacion ] ];
19
20     salida += "Calificacion\tFrecuencia\n";
21
22     for ( int calificacion = 1;
23           calificacion < frecuencia.length;
24           calificacion++ )
25         salida += calificacion + "\t" + frecuencia[ calificacion ] + "\n";
26
27     JTextArea areaSalida = new JTextArea( 11, 10 );
28     areaSalida.setText( salida );
29
30     JOptionPane.showMessageDialog( null, areaSalida,
31                               "Programa de encuesta a estudiantes",
32                               JOptionPane.INFORMATION_MESSAGE );
33
34     System.exit( 0 );
35 } // fin de main
36 } // fin de la clase EncEstudiantes

```

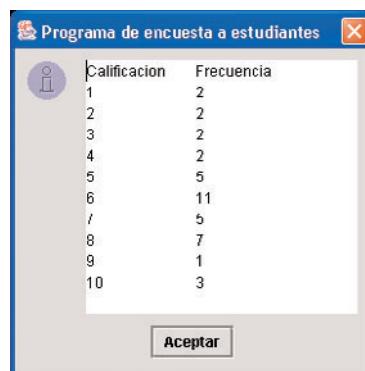


Figura 25.19 Un programa de análisis de una encuesta sencilla a estudiantes.

para contar el número de ocurrencias de cada respuesta. Ignoramos el primer elemento, `frecuencia[0]`, ya que es más lógico hacer que la respuesta 1 incremente a `frecuencia[1]`, en lugar de a `frecuencia[0]`. Esto nos permite utilizar cada respuesta de manera directa como un subíndice del arreglo `frecuencia`. Cada elemento del arreglo se utiliza como un contador para una de las respuestas de la encuesta.

Buena práctica de programación 25.7



Esfuérzese en favor de la claridad de sus programas. Algunas veces vale la pena sacrificar el uso más eficiente de la memoria o el tiempo de procesamiento, a favor de escribir programas más claros.

Tip de rendimiento 25.2



Algunas veces las consideraciones de rendimiento se contraponen a las consideraciones de claridad.

El ciclo **for** (líneas 15 a 18) toma las respuestas, una a la vez, del arreglo **respuestas** e incrementa uno de los 10 contadores del arreglo **frecuencia** (**frecuencia[1]** a **frecuencia[10]**). La instrucción clave del ciclo es

```
++frecuencia[ respuestas[ contestacion ] ];
```

Esta instrucción incrementa el contador **frecuencia** apropiado, de acuerdo con el valor de **respuestas[contestacion]**.

Consideremos diversas iteraciones del ciclo **for**. Cuando el contador **contestacion** es **0**, **respuestas[contestacion]** es el valor del primer elemento del arreglo **respuestas** (es decir, **1**), por lo que **++frecuencia[respuestas[contestacion]]**; en realidad se interpreta como

```
++frecuencia[ 1 ];
```

la cual incrementa el elemento uno del arreglo. Al evaluar la expresión, comience con el valor que se encuentra en el conjunto de corchetes más internos (**contestacion**). Una vez que sepa el valor de **contestacion**, conecte el valor que se encuentra en la expresión y evalúe el siguiente conjunto externo de corchetes (**respuestas[contestacion]**). Después utilice ese valor como el subíndice del arreglo **frecuencia** para determinar cuál contador incrementar.

Cuando **contestacion** es **1**, **respuestas[contestacion]** es el valor del segundo elemento del arreglo **respuestas** (es decir, **2**), por lo que **++frecuencia[respuestas[contestacion]]**; en realidad se interpreta como

```
++frecuencia[ 2 ];
```

la cual incrementa el elemento dos del arreglo (el tercer elemento del arreglo).

Cuando **contestacion** es **2**, **respuestas[contestacion]** es el valor del tercer elemento del arreglo **respuestas** (es decir, **6**), por lo que **++frecuencia[respuestas[contestacion]]**; en realidad se interpreta como

```
++frecuencia[ 6 ];
```

la cual incrementa el elemento seis del arreglo (el séptimo elemento del arreglo), y así sucesivamente. Observe que independientemente del número de respuestas procesadas de la encuesta, sólo se requiere un arreglo de 11 elementos (sin tomar en cuenta el elemento cero) para resumir los resultados, ya que todos los valores de respuesta se encuentran entre 1 y 10, y los valores de los subíndices para un arreglo de 11 elementos van de 0 a 10. También observe que los resultados son correctos, debido a que los elementos del arreglo **frecuencia** se inicializaron automáticamente en cero cuando el arreglo se asignó con **new**.

Si los datos contuvieran valores inválidos, como **13**, el programa intentaría sumar **1** a **frecuencia[13]**, lo que estaría fuera de los límites del arreglo. En C y en C++, el compilador permitiría tales referencias, y en tiempo de ejecución, el programa sobre pasaría el final del arreglo hacia donde creyera que se encuentra el elemento número 13, y sumaría 1 a lo que estuviera en esa ubicación de memoria. Esto podría potencialmente modificar otra variable del programa, o incluso podría resultar en una terminación prematura del programa. Java proporciona mecanismos para evitar el acceso a elementos que se encuentren fuera de los límites de un arreglo.

Tip para prevenir errores 25.2



Cuando se ejecuta un programa en Java, el intérprete de Java verifica los subíndices de los elementos del arreglo para asegurarse de que son válidos (es decir, todos los subíndices deben ser mayores o iguales que 0, y menores que la longitud del arreglo). Si hay un subíndice inválido, Java genera una excepción.



Tip para prevenir errores 25.3

Las excepciones se utilizan para indicar que ocurrió un error en un programa. Éstas permiten que el programador se recupere del error y continúe con la ejecución del programa, en lugar de terminarlo de manera anormal. Cuando se hace una referencia inválida hacia un arreglo, se genera una excepción `ArrayIndexOutOfBoundsException`.



Error común de programación 25.16

Hacer referencia a un elemento que se encuentra fuera de los límites de un arreglo, es un error lógico.



Tip para prevenir errores 25.4

Cuando se hace un ciclo a través de un arreglo, los subíndices de éste nunca deben ir por debajo de 0 y siempre deben ser menores que el número total de elementos del arreglo (uno menos que el tamaño de éste). Asegúrese de que la condición de terminación del ciclo evite el acceso a elementos fuera de este rango.



Tip para prevenir errores 25.5

Los programas deben validar que todos los valores de entrada sean correctos para prevenir que información errónea afecte a los cálculos de un programa.

Nuestra siguiente aplicación (figura 25.20) lee los números de un arreglo y grafica la información en un gráfico de barras (o histograma); cada número se imprime, y después, a un lado de éstos, se despliega una barra consistente en los asteriscos que representen a ese número. El ciclo anidado **for** (líneas 13 a 18) agrega las barras a la **String** que se desplegará en el **JTextArea areaSalida** de un diálogo de mensaje. Observe la condición de terminación de ciclo de la estructura **for** interna de la línea 16 (**j <= n[i]**). Cada vez que se alcanza la estructura **for** interna, ésta cuenta de 1 hasta **n[i]**, por lo que utiliza un valor del arreglo **n** para determinar el valor final de la variable de control **j** y el número de asteriscos a desplegar.

```

1 // Figura 25.20: Histograma.java
2 // Programa de impresión de un histograma
3 import javax.swing.*;
4
5 public class Histograma {
6     public static void main( String args[] )
7     {
8         int n[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9         String salida = "";
10
11         salida += "Elemento\tValor\tHistograma";
12
13         for ( int i = 0; i < n.length; i++ ) {
14             salida += "\n" + i + "\t" + n[ i ] + "\t";
15
16             for ( int j = 1; j <= n[ i ]; j++ ) // imprime una barra
17                 salida += "*";
18         } // fin de for
19
20         JTextArea areaSalida = new JTextArea( 11, 30 );
21         areaSalida.setText( salida );
22
23         JOptionPane.showMessageDialog( null, areaSalida,
24             "Programa de impresión de un histograma",
25             JOptionPane.INFORMATION_MESSAGE );
26
27         System.exit( 0 );
28     } // fin de main
29 } // fin de la clase Histograma

```

Figura 25.20 Un programa que imprime histogramas. (Parte 1 de 2.)



Figura 25.20 Un programa que imprime histogramas. (Parte 2 de 2.)

La sección 25.6 indicó que existe un método más elegante para escribir el programa de tiro de dados de la figura 25.12. El programa tiraba 6000 veces un dado de seis lados. Una versión con arreglos de esta aplicación aparece en la figura 25.21. Las líneas 16 a 35 de la figura 25.12 se reemplazan con la línea 13 de este programa, la cual utiliza el valor aleatorio **cara** como el subíndice del arreglo **frecuencia**, para determinar cuál elemento debe incrementarse durante cada iteración del ciclo. El cálculo del número aleatorio de la línea 12 produce números entre 1 y 6 (los valores del dado de seis lados), por lo que el arreglo **frecuencia** debe ser lo suficientemente grande para permitir valores de subíndices de 1 a 6. El número más pequeño de elementos requerido para un arreglo que tenga estos valores de subíndices es de siete elementos (valores de subíndices de 0 a 6). En este programa, ignoramos el elemento 0 del arreglo **frecuencia**. Además, las líneas 18 y 19 de este programa reemplazan a las líneas 40 a 47 de la figura 25.12. Podemos realizar un ciclo a través del arreglo **frecuencia**, por lo que no tenemos que numerar cada línea de texto a desplegar en el **JTextArea**, como hicimos en la figura 25.12.

```

1 // Figura 25.21: TiraDado.java
2 // Tira los dados 6000 veces
3 import javax.swing.*;
4
5 public class TiraDado {
6     public static void main( String args[] )
7     {
8         int cara, frecuencia[] = new int[ 7 ];
9         String salida = "";
10
11        for ( int tiro = 1; tiro <= 6000; tiro++ ) {
12            cara = 1 + ( int ) ( Math.random() * 6 );
13            ++frecuencia[ cara ];
14        }
15
16        salida += "Cara\tFrecuencia";
17
18        for ( cara = 1; cara < frecuencia.length; cara++ )
19            salida += "\n" + cara + "\t" + frecuencia[ cara ];
20
21        JTextArea areaSalida = new JTextArea( 7, 10 );
22        areaSalida.setText( salida );
23
24        JOptionPane.showMessageDialog( null, areaSalida,
25            "Tira los dados 6000 veces",
26            JOptionPane.INFORMATION_MESSAGE );

```

Figura 25.21 Programa para tirar dados por medio de arreglos, en lugar de la instrucción **switch**. (Parte 1 de 2.)

```

27
28     System.exit( 0 );
29 } // fin de main
30 } // fin de la clase TiraDado

```

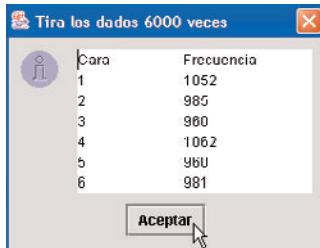


Figura 25.21 Programa para tirar dados por medio de arreglos, en lugar de la instrucción **switch**.
(Parte 2 de 2.)

25.11 Referencias y parámetros de referencias

Dos formas para pasar argumentos hacia los métodos (o funciones) en muchos lenguajes de programación (como C y C++) son las *llamadas por valor* y las *llamadas por referencia* (también conocidas como *pasar por valor* y *pasar por referencia*). Cuando un argumento se pasa mediante una llamada por valor, se hace una *copia* del valor del argumento y se pasa al método llamado.



Tip para prevenir errores 25.6

Con una llamada por valor, las modificaciones a la copia del método llamado no afecta el valor original de la variable en el método que llama. Esto evita los efectos colaterales accidentales que afectan grandemente el desarrollo de sistemas de software confiables.

Con una llamada por referencia, quien realiza la llamada proporciona al método llamado la habilidad de acceder directamente a los datos de quien llama y de modificar esos datos, si el método llamado lo elige así. Las llamadas por referencia pueden mejorar el rendimiento, ya que eliminan la sobrecarga de copiar grandes cantidades de datos, pero pueden debilitar la seguridad, ya que el método llamado puede acceder a los datos de quien lo llamó.



Observación de ingeniería de software 25.5

A diferencia de otros lenguajes, Java no permite al programador elegir si pasa cada argumento por medio de una llamada por valor o por medio de una llamada por referencia. Las variables de tipos de datos primitivos siempre se pasan por valor. Los objetos no se pasan hacia métodos; en su lugar, se pasan hacia los métodos las referencias a objetos. Las referencias mismas también se pasan por valor. Cuando un método recibe una referencia a un objeto, el método puede manipular directamente al objeto.



Observación de ingeniería de software 25.6

Cuando se devuelve información desde un método a través de una instrucción **return**, las variables de tipos de datos primitivos siempre se devuelven por valor (es decir, se devuelve una copia), y los objetos siempre se devuelven por referencia (es decir, se devuelve una referencia al objeto).

Para pasar una referencia a un objeto hacia un método, simplemente especifique en la llamada al método el nombre de la referencia. Al mencionar la referencia por medio del nombre de su parámetro en el cuerpo del método llamado, en realidad se hace referencia al objeto original en memoria, y se puede acceder directamente al objeto original por medio del método llamado.

Java trata a los arreglos como objetos, por lo que éstos se pasan hacia los métodos por medio de una llamada por referencia; un método llamado puede acceder a los elementos de los arreglos originales de quien le llamó. El nombre de un arreglo en realidad es una referencia a un objeto que contiene los elementos del arreglo y la variable de instancia **length**, la cual indica el número de elementos en el arreglo. En la siguiente sección demostraremos las llamadas por valor y las llamadas por referencia, utilizando arreglos.

Tip de rendimiento 25.3



Pasar arreglos por referencia tiene sentido por razones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Por mucho, pasar con frecuencia arreglos implicaría una pérdida de tiempo y de almacenamiento para las copias de los arreglos.

25.12 Arreglos con múltiples subíndices

Los arreglos con múltiples subíndices (con dos subíndices) con frecuencia se utilizan para representar *tablas* de valores que consisten en información acomodada en *filas* y *columnas*. Para identificar un elemento en particular de una tabla, debemos especificar los dos subíndices; por convención, el primero identifica la fila del elemento, y el segundo identifica a la columna. Los arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como *arreglos con dos subíndices*. Observe que los arreglos con múltiples subíndices pueden tener más de dos subíndices. Java no soporta directamente los arreglos con múltiples subíndices, pero permite al programador especificar arreglos con un solo subíndice, cuyos elementos también son arreglos con un solo subíndice, con lo que se logra el mismo efecto. La figura 25.22 ilustra un arreglo con dos subíndices, **a**, que contiene tres filas y cuatro columnas (es decir, un arreglo de 3 por 4). En general, a un arreglo con *m* filas y *n* columnas se le conoce como arreglo de *m* por *n*.

Cada elemento del arreglo **a** se identifica en la figura 25.22, por medio de un nombre de elemento de la forma **a[i][j]**; **a** es el nombre del arreglo e **i** y **j** son los subíndices que identifican de manera única a la fila y a la columna de cada elemento en **a**. Observe que los nombres de los elementos en la primera fila tienen un primer subíndice **0**; los nombres de los elementos en la cuarta columna tienen un segundo subíndice **3**.

Los arreglos con múltiples subíndices pueden inicializarse en declaraciones como un arreglo con un solo subíndice. Un arreglo con dos subíndices **b[2][2]** podría declararse e inicializarse con

```
int b[][] = { { 1, 2 }, { 3, 4 } };
```

Los valores se agrupan por fila, entre llaves. Entonces, **1** y **2** inicializan **b[0][0]** y **b[0][1]**, y **3** y **4** inicializan **b[1][0]** y **b[1][1]**. El compilador determina el número de filas, contando el número de listas de subinicialización (representadas por conjuntos de llaves) en la lista principal de inicialización. El compilador determina el número de columnas en cada fila, contando el número de valores inicializadores en la lista de subinicialización para esa fila.

Los arreglos con múltiples subíndices se mantienen como arreglos de arreglos. La declaración

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

crea un arreglo entero **b** con la fila **0** que contiene dos elementos (**1** y **2**), y la fila **1** que contiene tres elementos (**3**, **4** y **5**).

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Fila 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Fila 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

↑ ↑ ↑ ↑
 Subíndice de columna
 ↓ ↓ ↓ ↓
 Subíndice de fila
 ↓
 Nombre del arreglo

Figura 25.22 Un arreglo con dos subíndices que consta de tres filas y cuatro columnas.

Un arreglo con múltiples subíndices con el mismo número de columnas en cada fila puede asignarse dinámicamente. Por ejemplo, un arreglo de 3 por 3 se asigna de la siguiente manera:

```
int b[][];
b = new int[ 3 ][ 3 ];
```

Así como con los arreglos de un solo subíndice, los elementos de un arreglo con dos subíndices se inicializan cuando **new** crea el objeto arreglo.

Un arreglo con múltiples subíndices en el que cada fila tiene un número diferente de columnas, puede asignarse dinámicamente de la siguiente manera:

```
int b[][];
b = new int[ 2 ][ ];
b[ 0 ] = new int[ 5 ];      //asigna las columnas de la fila 0
b[ 1 ] = new int[ 3 ];      //asigna las columnas de la fila 1
```

El código anterior crea un arreglo bidimensional con dos filas. La fila 0 tiene cinco columnas y la fila 1 tiene tres.

El applet de la figura 25.23 muestra la inicialización de arreglos con dos subíndices en las declaraciones, y utiliza ciclos **for** anidados para recorrer los arreglos (es decir, para manipular cada elemento del arreglo).

```

1 // Figura 25.23: InicArreglo.java
2 // Inicialización de Arreglos multidimensionales
3 import java.awt.Container;
4 import javax.swing.*;
5
6 public class InicArreglo extends JApplet {
7     JTextArea areaSalida;
8
9     // inicializa el objeto
10    public void init()
11    {
12        areaSalida = new JTextArea();
13        Container c = getContentPane();
14        c.add( areaSalida );
15
16        int arreglo1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
17        int arreglo2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
18
19        areaSalida.setText( "Los Valores en arreglo1 por fila son\n" );
20        construyeSalida( arreglo1 );
21
22        areaSalida.append( "\nLos Valores en arreglo2 por fila son\n" );
23        construyeSalida( arreglo2 );
24    } // fin del método init
25
26    public void construyeSalida( int a[][] )
27    {
28        for ( int i = 0; i < a.length; i++ ) {
29
30            for ( int j = 0; j < a[ i ].length; j++ )
31                areaSalida.append( a[ i ][ j ] + " " );
32
33            areaSalida.append( "\n" );
34        } // fin de for
35    } // fin del método construyeSalida
36 } // fin de la clase InicArreglo
```

Figura 25.23 Inicialización de arreglos multidimensionales. (Parte 1 de 2.)

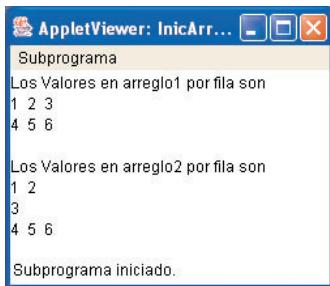


Figura 25.23 Inicialización de arreglos multidimensionales. (Parte 2 de 2.)

El programa declara dos arreglos en el método **init**. La declaración de **arreglo1** (línea 16) proporciona seis inicializadores en dos sublistas. La primera sublista inicializa la primera fila del arreglo con los valores **1**, **2** y **3**; y la segunda sublista inicializa la segunda fila del arreglo con los valores **4**, **5** y **6**. La declaración de **arreglo2** (línea 17) proporciona seis inicializadores en tres sublistas. La sublista para la primera fila inicializa explícitamente la primera fila para que tenga dos elementos con los valores **1** y **2**, respectivamente. La sublista para la segunda fila inicializa la segunda fila para que tenga un elemento con el valor **3**. La sublista para la tercera fila inicializa la tercera fila con los valores **4**, **5** y **6**.

El método **init** llama al método **construyeSalida** de las líneas 20 y 23 para agregar los elementos de cada arreglo al **JTextArea areaSalida**. La definición del método **construyeSalida** especifica el parámetro del arreglo como **int a[][]** para indicar que un arreglo con dos subíndices se recibirá como argumento. Observe el uso de una estructura **for** anidada para desplegar las filas de cada arreglo con dos subíndices. En la estructura **for** externa, la expresión **a.lengtht** determina el número de filas en el arreglo. En la estructura **for** interna, la expresión **a[i].lengtht** determina el número de columnas en cada fila del arreglo. Esta condición permite al ciclo determinar, para cada fila, el número exacto de columnas.

Muchas manipulaciones comunes de arreglos utilizan estructuras de repetición **for**. Por ejemplo, la siguiente estructura **for** establece en cero a todos los elementos de la tercera fila del arreglo **a** correspondiente a la figura 25.22:

```
for ( int col = 0; col < a[ 2 ].lengtht; col++ )
    a[ 2 ][ col ] = 0;
```

Nosotros especificamos la *tercera fila*, por lo tanto, sabemos que el primer subíndice siempre es **2** (**0** es la primera fila, y **1** es la segunda). El ciclo **for** varía sólo el segundo subíndice (es decir, el subíndice de columna). La estructura **for** anterior es equivalente a las instrucciones de asignación

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente estructura **for** anidada determina el total de todos los elementos del arreglo **a**:

```
int total = 0;

for ( int fila = 0; fila < a.lengtht; fila++ )
    for ( int col = 0; col < a[ fila ].lengtht; col++ )
        total += a[ fila ][ col ];
```

La estructura **for** totaliza los elementos del arreglo, una fila a la vez. La estructura **for** externa comienza estableciendo el subíndice de **fila** en **0**, por lo que los elementos de la primera fila pueden ser totalizados por la estructura **for** interna. La estructura **for** externa después incrementa en **1** a fila, por lo que la segunda fila puede ser totalizada. Después, la estructura **for** externa incrementa en **2** a fila, por lo que la tercera fila puede ser totalizada. El resultado puede desplegarse cuando la estructura **for** anidada termina.

RESUMEN

- Los tipos primitivos (**boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**) son los bloques de construcción para tipos más complicados en Java.
- Java requiere que todas las variables tengan un tipo, antes de que puedan utilizarse en un programa. Por esta razón, Java se conoce como un lenguaje fuertemente basado en tipos.
- Los tipos primitivos en Java son portables a través de todas las plataformas de cómputo que soportan Java.
- Java utiliza estándares reconocidos internacionalmente para formatos de caracteres (Unicode) y de números de punto flotante (IEEE 754).
- A las variables de tipos **char**, **byte**, **short**, **int**, **long**, **float** y **double** se les da el valor de **0**, de manera predeterminada, y a las de tipo **boolean** se les da el valor de **false**, también de manera predeterminada.
- Los operadores lógicos pueden utilizarse para formar condiciones complejas, combinando condiciones. Los operadores lógicos son **&&**, **&**, **||**, **|**, **^** y **!**, los cuales significan AND lógico, AND lógico booleano, OR lógico, OR lógico booleano incluyente, OR lógico booleano excluyente y NOT lógico (negación), respectivamente.
- La mejor forma de desarrollar y mantener un programa grande es dividirlo en diversos módulos de programas pequeños, los cuales son más manejables que el programa original. Los módulos se escriben en Java como clases y métodos.
- El área en donde se despliega un **JApplet** en la pantalla tiene un panel de contenido al que los componentes GUI deben adjuntarse para que puedan desplegarse en tiempo de ejecución. El panel de contenido es un objeto de la clase **Container** del paquete **java.awt**.
- El método **getContentPane** devuelve una referencia hacia el panel de contenido del applet.
- El formato general para la definición de un método es

tipo del valor de retorno nombre del método(lista de parámetros)
 {
declaraciones e instrucciones
 }

El *tipo del valor de retorno* establece el tipo del valor devuelto hacia el método que realiza la llamada. Si un método no devuelve un valor, el *tipo del valor de retorno* es **void**. El *nombre del método* es cualquier identificador válido. La *lista de parámetros* es una lista separada por comas que contiene las declaraciones de las variables que se pasarán al método. Si un método no recibe valor alguno, la *lista de parámetros* está vacía. El cuerpo del método es el conjunto de *declaraciones e instrucciones* que constituyen el método.

- Una lista de parámetros vacía se especifica con paréntesis vacíos.
- Los argumentos pasados a un método deben coincidir en número, tipo y orden con los parámetros en la definición del método.
- Cuando un programa encuentra un método, el control se transfiere del punto de invocación hacia el método llamado, el método se ejecuta, y el control regresa a quien hizo la llamada.
- Un método llamado puede devolver el control hacia quien hizo la llamada, de tres formas. Si el método no devuelve un valor, el control se devuelve cuando se alcanza la llave derecha del final del método, o ejecutando la instrucción

return;

- Si el método devuelve un valor, la instrucción

return expresion;

devuelve el valor de *expresión*.

- El método **Math.random** genera un valor **double** mayor o igual que 0.0, pero menor que 1.0.
- Los valores producidos por **Math.random** pueden escalarse y desplazarse para producir valores en un rango en particular.
- La ecuación general para escalar y desplazar un número aleatorio es

n = a + (int) (Math.random() * b);

donde **a** es el valor de desplazamiento (el primer número del rango deseado de enteros consecutivos) y **b** es el factor de escalamiento (el ancho del rango deseado de enteros consecutivos).

- Una clase puede heredar atributos y comportamientos existentes (datos y métodos) de otra clase especificada a la derecha de la palabra reservada **extends** en la definición de la clase. Además, una clase puede implementar una o más interfaces. Una interfaz especifica uno o más comportamientos (es decir, métodos) que usted puede definir en la definición de una clase.

- La interfaz **ActionListener** especifica que esta clase debe definir un método con la primera línea

```
public void actionPerformed( ActionEvent e )
```

- La tarea del método **actionPerformed** es la de procesar una interacción de usuario con un componente GUI que genera un evento de acción. Este método es llamado automáticamente, en respuesta a la interacción del usuario. A este proceso se le conoce como manejo de eventos. El evento es la interacción de usuario (oprimir el botón). El manejador de eventos es el método **actionPerformed**, el cual es llamado automáticamente en respuesta al evento. A este estilo de programación se le conoce como programación manejada por eventos.
- La palabra reservada **final** se utiliza para declarar variables constantes. Las variables constantes deben inicializarse una vez antes de utilizarlas, y no pueden modificarse después. Las constantes variables también se conocen como constantes nombradas o variables de sólo lectura.
- Una **JLabel** contiene una cadena de caracteres a desplegar en la pantalla. Normalmente, una **JLabel** indica el propósito de otro elemento de la interfaz gráfica de usuario en la pantalla.
- Los **JTextFields** se utilizan para obtener información desde el teclado o para desplegar información en la pantalla.
- Cuando el usuario oprime un **JButton**, normalmente el programa responde realizando una tarea.
- El método **setLayout** de **Container** define el administrador de diseño para la interfaz de usuario del applet. Los administradores de diseño se proporcionan para acomodar los componentes GUI en un **Container**, para efectos de presentación. Los administradores de diseño proporcionan las capacidades básicas de diseño que determinan la posición y el tamaño de cada componente GUI adjunto al contenedor. Esto permite al programador concentrarse en la “apariencia visual” básica, y deja a los administradores de diseño el procesamiento de la mayoría de los detalles de diseño.
- **FlowLayout** es el administrador de diseño más básico. Los componentes GUI se colocan en un **Container** de izquierda a derecha, en el orden en el que se adjuntan al **Container** por medio del método **add**. Cuando se alcanza el borde del contenedor, los componentes continúan en la siguiente línea.
- Antes de que pueda procesarse cualquier evento, cada componente GUI debe saber cuál objeto del programa define el método de manipulación de eventos que será llamado cuando ocurra un evento. El método **addActionListener** se utiliza para indicarle a un **JButton** que otro objeto está escuchando los eventos de acción y define el método **actionPerformed**. A esto se le llama registro del manipulador de eventos con el componente GUI (también quisieramos llamarlo la línea que *empieza a escuchar*, ya que el applet ahora está escuchando los eventos del botón). Para responder a un evento de acción, debemos definir una clase que implemente un **ActionListener** (esto requiere que la clase también defina un método **actionPerformed**) y debemos registrar el manipulador de eventos con el componente GUI.
- El método **showStatus** recibe un argumento **String** y lo despliega en la barra de estado del **appletviewer** o navegador.
- El **appletviewer** o el navegador llama una vez al método **init** de un **applet**, cuando éste se carga para su ejecución. Este método realiza la inicialización de un applet. El método **start** del applet es llamado después de que el método **init** completa su ejecución y cada vez que el usuario del navegador regresa a la página HTML en donde reside el applet (después de explorar otra página HTML).
- El método **paint** es llamado después de que el método **init** completa su ejecución y una vez que el método **start** ha iniciado su ejecución para dibujar en el applet. También se le llama automáticamente cada vez que el applet necesita repintarse.
- El método **stop** es llamado cuando el applet debe suspender su ejecución; normalmente cuando el usuario del navegador abandona la página HTML en donde el applet reside.
- El método **destroy** de un applet es llamado cuando el applet está siendo removido de la memoria; normalmente cuando el usuario del navegador abandona la sesión de navegación.
- El método **repaint** puede ser llamado en un applet para ocasionar una llamada fresca a **paint**. El método **repaint** invoca a otro método llamado **update**, y le pasa el objeto **Graphics**. El método **update** borra cualquier dibujo que se haya hecho previamente en el applet, después invoca al método **paint**, y le pasa el objeto **Graphics**.
- Cuando se declara un arreglo, el tipo del arreglo y los corchetes pueden combinarse al principio de la declaración, para indicar que todos los identificadores de la declaración representan arreglos, como en

```
double[] arreglo1, arreglo2;
```

- Los elementos de un arreglo pueden inicializarse por declaración (utilizando listas de inicializadores), por asignación y por entrada.
- Java evita las referencias a los elementos que se encuentren más allá de los límites de un arreglo.
- Para pasar un arreglo a un método, se pasa el nombre del arreglo. Para pasar un solo elemento de un arreglo a un método, simplemente pase el nombre del arreglo, seguido por el subíndice (contenido entre corchetes) del elemento en particular.
- Los arreglos pasan a los métodos por medio de una llamada por referencia; por lo tanto, los métodos llamados pueden modificar los valores de los elementos en los arreglos originales de quien hace la llamada. Los elementos de tipos de datos primitivos correspondientes a un arreglo son pasados a los métodos por medio de llamadas por valor.
- Los arreglos pueden utilizarse para representar tablas de valores que consisten en información ordenada en filas y columnas. Para identificar un elemento particular de la tabla, se especifican dos subíndices: el primero identifica la fila en la que se encuentra el elemento, y el segundo la columna. Las tablas o arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como arreglos con dos subíndices.
- Un arreglo con dos subíndices puede inicializarse con una lista de inicializadores de la forma

tipoArreglo nombreArreglo[][] = { { fila1 sublistा }, { fila2, sublistा }, ... };

- Para crear dinámicamente un arreglo con un número fijo de filas y columnas, utilice

tipoArreglo nombreArreglo[][] = new tipoArreglo[numFilas] [numColumnas];

- Para pasar una fila de un arreglo con dos subíndices a un método que recibe un arreglo con un solo subíndice, simplemente pase el nombre del arreglo seguido por el subíndice de fila.

TERMINOLOGÍA

a[i]	cuadro de desplazamiento	long
a[i][j]	declaración de un método	Math.E
AND lógico (&&)	declarar un arreglo	Math.PI
AND lógico booleano (&)	definición de un método	método
API de Java (biblioteca de clases de Java)	desplazamiento	método actionPerformed
argumento en una llamada a un método	devolver	método append de la clase JTextArea
arreglo	divide y vencerás	método definido por el programador
arreglo con dos subíndices	double	método destroy de JApplet
arreglo con múltiples subíndices	duración	método init de JApplet
arreglo con un solo subíndice	efectos colaterales	método llamado
arreglo de <i>m</i> por <i>n</i>	elemento cero	método Math.random
barra de desplazamiento	elemento de probabilidad	método paint de JApplet
break	elemento de un arreglo	método que llama
clase	error de desplazamiento en uno	método repaint de JApplet
clase ActionEvent	escalar	método setFont
clase FlowLayout	evaluación de cortocircuito	método setLayout de JApplet
clase Font de java.awt	expresión de tipo mixto	método showStatus de JApplet
clase JButton del paquete javax.swing	final	JApplet
clase JLabel del paquete javax.swing	Font.BOLD	método start de JApplet
clase JScrollPane	Font.ITALIC	método stop de JApplet
clase JTextArea	Font.PLAIN	método update de JApplet
clase JTextField del paquete javax.swing	formato tabular	métodos de la clase Math
conjunto de caracteres ISO Unicode	generación de números aleatorios	negación lógica (!)
constante nombrada	ingeniería de software	nombre de un arreglo
corchetes []	inicialización de un arreglo	operador !
	inicializador	operador &&
	interfaz ActionListener	operador
	invocar a un método	operador de llamada a un método, ()
	lista de inicialización de arreglos	
	llamada a un método	

operador unario	paso por valor	subíndice de columna
operadores lógicos	programa modular	subíndice de fila
OR lógico ()	pulgar de una barra de desplazamiento	tabla de valores
OR lógico booleano incluyente ()	reutilización de software	tipo de valor de retorno
OR lógico booleano excluyente (^)	simulación	tipos de referencias
parámetro de referencia	sobrecarga de métodos	valor de un elemento
paso de arreglos a métodos	subíndice	verificación de límites
paso por referencia		void

ERRORES COMUNES DE PROGRAMACIÓN

- 25.1 Utilizar una palabra reservada como identificador, es un error de sintaxis.
- 25.2 En expresiones que utilizan el operador **&&**, es posible que una condición (a la que llamaremos condición dependiente) requiera de otra condición para ser **true**, de tal modo que ésta tenga sentido al evaluar la condición dependiente. En este caso, la condición dependiente debe colocarse después de la otra condición, o es posible que ocurra un error.
- 25.3 Definir un método fuera de las llaves correspondientes a la definición de una clase, es un error de sintaxis.
- 25.4 Omitir el tipo del valor de retorno en la definición de un método, es un error de sintaxis.
- 25.5 Olvidar devolver un valor por parte de un método que se supone debe hacerlo, es un error de sintaxis. Si se especifica un tipo de valor de retorno diferente de **void**, el método debe contener una instrucción **return**.
- 25.6 Devolver un valor desde un método, cuyo tipo de retorno se declaró como **void**, es un error de sintaxis.
- 25.7 Declarar parámetros del mismo tipo en un método, como **float x, y**, en lugar de **float x, float y**, es un error de sintaxis, ya que se necesitan tipos para cada parámetro de la lista de parámetros.
- 25.8 Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de una definición de método, es un error de sintaxis.
- 25.9 Redefinir un parámetro de un método como una variable local del método, es un error de sintaxis.
- 25.10 Pasar un método a un argumento que no es compatible con el tipo correspondiente al parámetro, es un error de sintaxis.
- 25.11 Definir un método dentro de otro, es un error de sintaxis.
- 25.12 Despues de que se inicializó una variable **final**, intentar asignar otro valor a esa variable es un error de sintaxis.
- 25.13 Proporcionar una definición para uno de los métodos de **JApplet init, start, paint, stop o destroy**, que no coincide con los encabezados de los métodos que muestra la figura 25.14, dará como resultado un método que no será llamado automáticamente durante la ejecución del applet.
- 25.14 A diferencia de C o de C++, el número de elementos en el arreglo nunca se especifica entre corchetes después del nombre del arreglo en una declaración. La declaración **int c[12];** ocasiona un error de sintaxis.
- 25.15 Asignar un valor a una constante variable después de inicializarla, es un error de sintaxis.
- 25.16 Hacer referencia a un elemento que se encuentra fuera de los límites de un arreglo, es un error lógico.

TIPS PARA PREVENIR ERRORES

- 25.1 Los métodos pequeños son más fáciles de probar, depurar y comprender, que aquellos que son grandes.
- 25.2 Cuando se ejecuta un programa en Java, el intérprete de Java verifica los subíndices de los elementos del arreglo para asegurarse de que son válidos (es decir, todos los subíndices deben ser mayores o iguales que 0, y menores que la longitud del arreglo). Si hay un subíndice inválido, Java genera una excepción.
- 25.3 Las excepciones se utilizan para indicar que ocurrió un error en un programa. Éstas permiten que el programador se recupere del error y continúe con la ejecución del programa, en lugar de terminarlo de manera anormal. Cuando se hace una referencia inválida hacia un arreglo, se genera una excepción **ArrayIndexOutOfBoundsException**.
- 25.4 Cuando se hace un ciclo a través de un arreglo, los subíndices de éste nunca deben ir por debajo de 0 y siempre deben ser menores que el número total de elementos del arreglo (uno menos que el tamaño de éste). Asegúrese de que la condición de terminación del ciclo evite el acceso a elementos fuera de este rango.

- 25.5** Los programas deben validar que todos los valores de entrada sean correctos para prevenir que información errónea afecte a los cálculos de un programa.
- 25.6** Con una llamada por valor, las modificaciones a la copia del método llamado no afecta el valor original de la variable en el método que llama. Esto evita los efectos colaterales accidentales que afectan grandemente el desarrollo de sistemas de software confiables.

BUENAS PRÁCTICAS DE PROGRAMACIÓN

- 25.1** Por claridad, evite expresiones con efectos colaterales en las condiciones. Los efectos colaterales pueden parecer convenientes, pero con frecuencia representan más problemas que ventajas.
- 25.2** Coloque una línea en blanco entre las definiciones de métodos para separarlos y para mejorar la legibilidad del programa.
- 25.3** Aunque no es incorrecto hacerlo, en la definición de un método no utilice los mismos nombres para los argumentos pasados a él y para los parámetros correspondientes. Esto ayuda a evitar la ambigüedad.
- 25.4** Elegir nombres descriptivos para los métodos y para los parámetros hace que los programas sean más legibles, y ayuda a evitar el uso excesivo de comentarios.
- 25.5** Sólo utilice letras mayúsculas (con guiones bajos entre las palabras) en los nombres de variables **final**. Esto hace que las constantes resalten en un programa.
- 25.6** Utilizar variables **final** con nombres descriptivos, en lugar de utilizar constantes enteras (como 2), hace que los programas sean legibles.
- 25.7** Esfuérzese en favor de la claridad de sus programas. Algunas veces vale la pena sacrificar el uso más eficiente de la memoria o el tiempo de procesamiento, a favor de escribir programas más claros.

TIPS DE RENDIMIENTO

- 25.1** En expresiones que utilizan el operador **&&**, si las condiciones separadas son independientes una de la otra, haga que la condición que más probablemente sea falsa, se encuentre más a la izquierda. En expresiones que utilizan el operador **||**, haga que la condición que más probablemente sea verdadera, se encuentre más a la izquierda. Esto puede reducir el tiempo de ejecución de un programa.
- 25.2** Algunas veces las consideraciones de rendimiento se contraponen a las consideraciones de claridad.
- 25.3** Pasar arreglos por referencia tiene sentido por razones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Por mucho, pasar con frecuencia arreglos implicaría una pérdida de tiempo y de almacenamiento para las copias de los arreglos.

TIP DE PORTABILIDAD

- 25.1** Todos los tipos de datos primitivos en Java son portables, a través de todas las plataformas que soportan Java.

OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 25.1** Por lo general, un método no debe sobrepasar una página. Mejor aún, un método generalmente debe abarcar no más de media página. Independientemente de cuán largo sea un método, debe realizar bien una tarea. Los métodos pequeños promueven la reutilización de software.
- 25.2** Los programas deben escribirse como colecciones de métodos pequeños. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.
- 25.3** Es posible que un método que requiere un gran número de parámetros esté realizando demasiadas tareas. Considerar dividir el método en métodos más pequeños que realicen tareas separadas. Si es posible, el encabezado del método debe caber en una línea.
- 25.4** El encabezado de un método y las llamadas a él deben coincidir en número, tipo y orden de parámetros y argumentos.
- 25.5** A diferencia de otros lenguajes, Java no permite al programador elegir si pasa cada argumento por medio de una llamada por valor o por medio de una llamada por referencia. Las variables de tipos de datos primitivos siempre se pasan por valor. Los objetos no se pasan hacia métodos; en su lugar, se pasan hacia los métodos las referencias a

objetos. Las referencias mismas también se pasan por valor. Cuando un método recibe una referencia a un objeto, el método puede manipular directamente al objeto.

- 25.6** Cuando se devuelve información desde un método a través de una instrucción **return**, las variables de tipos de datos primitivos siempre se devuelven por valor (es decir, se devuelve una copia), y los objetos siempre se devuelven por referencia (es decir, se devuelve una referencia al objeto).

EJERCICIOS DE AUTOEVALUACIÓN

- 25.1** Complete los espacios en blanco:

- A los módulos de un programa en Java se les conoce como _____ y _____.
- A un método se le invoca con una _____.
- A una variable conocida sólo dentro del método en el que está definida se le conoce como _____.
- La instrucción _____ en un método llamado puede utilizarse para pasar el valor de una expresión de regreso hacia el método que hizo la llamada.
- La palabra reservada _____ se utiliza en el encabezado de un método para indicar que el método no devuelve valor alguno.
- Las tres formas de devolver el control desde un método llamada hasta el método que hizo la llamada son _____, _____ y _____.
- El método _____ se invoca una vez, cuando un applet comienza su ejecución.
- El método _____ se utiliza para producir números aleatorios.
- El método _____ se invoca cada vez que el usuario de un navegador vuelve a visitar la página HTML en la que reside el applet.
- El método _____ se invoca para dibujar en un applet.
- El método _____ invoca al método update del applet, el cual a su vez invoca al método paint del applet.
- El método _____ se invoca para un applet cada vez que el usuario de un navegador abandona la página HTML en la que reside el applet.
- El calificador _____ se utiliza para declarar variables de sólo lectura.

- 25.2** Proporcione el encabezado del método para cada uno de los siguientes:

- Método **hipotenusa**, el cual toma dos argumentos de punto flotante de doble precisión, **lado1** y **lado2**, y devuelve un resultado de punto flotante de doble precisión.
- Método **masPequenio**, el cual toma tres enteros, **x**, **y**, **z**, y devuelve un entero.
- Método **instrucciones**, el cual no toma argumentos y no devuelve valor alguno. [Nota: Tales métodos normalmente se utilizan para desplegar instrucciones para el usuario.]
- Método **intToFloat**, el cual toma un argumento entero, **numero**, y devuelve un resultado de punto flotante.

- 25.3** Encuentre el error en cada uno de los siguientes segmentos de programa, y explique cómo corregirlo:

- ```
int g() {
 System.out.println("Dentro del metodo g");
 int h() {
 System.out.println("Dentro del metodo h");
 }
}
```
- ```
int suma( int x, int y ) {
    int resultado ;
    resultado = x + y ;
}
```
- ```
int suma(int n) {
 if(n == 0)
 return 0 ;
 else
 n + suma(n - 1) ;
}
```
- ```
void f( float a ) {
    float a ;
    System.out.println( a ) ;
}
```

```

e) void product() {
    int a = 6, b = 5, c = 4, resultado ;
    resultado = a * b * c;
    System.out.println( "El resultado es " + resultado );
    return resultado;
}

```

- 25.4** Establezca si los siguientes enunciados son *verdaderos* o *falsos*. Si la respuesta es *falso*, explique por qué.
- Un arreglo puede almacenar muchos tipos diferentes de valores.
 - Un subíndice de arreglo normalmente debe ser del tipo de dato float.
 - Un elemento individual de un arreglo que se pasa a un método y se modifica en ese método contendrá el valor modificado cuando el método llamado complete su ejecución.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

- 25.1** a) Métodos y clases. b) Llamada a un método. c) Variable local. d) **return**. e) **void**. f) **return; o return expresion;** o al encontrar la llave derecha de cierre de método. g) **init**. h) **Math.random**. i) **start**. j) **paint**. k) **repaint**. l) **stop**. m) **final**.
- 25.2** a) **double hipotenusa(double lado1, double lado2)**
b) **int masPequenio(int x, int y, int z)**
c) **void instrucciones()**
d) **float intToFloat(int numero)**
- 25.3** a) Error: el método h está definido en el método g.
Corrección: mueva la definición de h fuera de la definición de g.
b) Error: se supone que el método debe devolver un entero, pero no lo hace.
Corrección: elimine la variable **resultado** y coloque la siguiente instrucción en el método:
- ```

return x + y;

```
- o agregue la siguiente instrucción al final del cuerpo del método:
- ```

return resultado;

```
- c) Error: el resultado de **n + suma(n - 1)** no es devuelto por este método recursivo, lo que ocasiona un error de sintaxis.
Corrección: escriba la instrucción en la cláusula **else** como **return n + suma(n - 1);**
d) Error: el punto y coma después del paréntesis derecho que encierra la lista de parámetros, y definir el parámetro **a** en la definición del método es incorrecto.
Corrección: elimine el punto y coma después del paréntesis derecho de la lista de parámetros, y elimine la declaración **float a;**
e) Error: el método devuelve un valor, cuando se supone que no debe hacerlo.
Corrección: cambie el tipo de retorno a **int**.
- 25.4** a) Falso. Un arreglo puede almacenar solamente valores del mismo tipo.
b) Falso. Un subíndice de arreglo debe ser un entero o una expresión entera.
c) Falso. Para elementos individuales de tipos primitivos de un arreglo, ya que éstos son pasados mediante una llamada por valor. Si se pasa una referencia a un arreglo, entonces las modificaciones a los elementos del arreglo se reflejan en el original. Además, un elemento individual de un tipo de clase pasado a un método, se pasa por medio de una llamada por referencia, y las modificaciones al objeto se reflejarán en el elemento original del arreglo.

EJERCICIOS

- 25.5** Responda cada una de las siguientes preguntas:
- ¿Qué significa elegir números “al azar”?
 - ¿Por qué el método **Math.random** es útil para simular juegos de azar?
 - ¿Por qué con frecuencia es necesario escalar y/o desplazar los valores producidos por **Math.random**?
 - ¿Por qué la simulación computarizada de situaciones reales es una técnica útil?

25.6 Escriba instrucciones que asigne enteros aleatorios a la variable `n` en los siguientes rangos:

- a) $1 \leq n \leq 2$
- b) $1 \leq n \leq 100$
- c) $0 \leq n \leq 9$
- d) $1000 \leq n \leq 1112$
- e) $-1 \leq n \leq 1$
- f) $-3 \leq n \leq 11$

25.7 Para cada uno de los siguientes conjuntos de enteros, escriba una sola instrucción que imprima un número al azar del conjunto.

- a) 2, 4, 6, 8, 10.
- b) 3, 5, 7, 9, 11.
- c) 6, 10, 14, 18, 22.

25.8 Defina un método `hipotenusa` que calcule la longitud de la hipotenusa de un triángulo recto, cuando se conocen los otros dos lados. El método debe tomar dos argumentos de tipo `double` y debe devolver la hipotenusa como un `double`. Incorpore este método a un applet que lea valores enteros para `lado1` y `lado2` desde `JTextFields`, y que realice el cálculo con el método `hipotenusa`. Determine la longitud de la hipotenusa para cada uno de los siguientes triángulos. [Nota: Registre la manipulación de eventos sólo en el segundo `JTextField`. El usuario debe interactuar con el programa, escribiendo los números en ambos `JTextFields` y oprimir *Entrar* en el segundo `JTextField`.]

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

25.9 Escriba un método `multiplo` que determine para un par de enteros, si el segundo es múltiplo del primero. El método debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero; de lo contrario, debe devolver `false`. Incorpore este método a un applet que introduzca una serie de pares de enteros (un par a la vez, utilizando `JTextFields`). [Nota: Registre la manipulación de eventos sólo en el segundo `JTextField`. El usuario debe interactuar con el programa, escribiendo los números en ambos `JTextFields` y oprimir *Entrar* en el segundo `JTextField`.]

25.10 Escriba un applet que introduzca enteros (uno a la vez) y que los pase, uno a la vez, hacia el método `esPar`, el cual utiliza el operador módulo para determinar si un entero es par. El método debe tomar un argumento entero y devolver `true` si el entero es par y `false` de lo contrario. Utilice un diálogo de entrada para obtener los datos del usuario.

25.11 Escriba un método `cuadradoDeAsteriscos` que despliegue un cuadrado sólido de asteriscos, cuyo lado se especifica en el parámetro entero `lado`. Por ejemplo, si `lado` es 4, el método despliega

```
*****
*****
*****
*****
```

Incorpore este método en un applet que lea un valor entero para `lado`, proporcionado desde el teclado, y que realice el dibujo por medio del método `cuadradoDeAsteriscos`. Observe que este método debe ser llamado desde el método `paint` del applet, y el objeto `Graphics` debe ser pasado desde `paint`.

25.12 Implemente los siguientes métodos de enteros:

- a) El método `celsius` devuelve el equivalente Celsius de la temperatura en Fahrenheit, por medio del cálculo

```
C = 5.0 / 9.0 * (F - 32);
```

- b) El método `fahrenheit` devuelve el equivalente Fahrenheit de la temperatura en Celsius.

```
F = 9.0 / 5.0 * C + 32;
```

- c) Utilice estos métodos para escribir un applet que permita al usuario introducir una temperatura en Fahrenheit y que se despliegue en Celsius, o que introduzca una temperatura en Celsius y que se despliegue en Fahrenheit.

[Nota: Este applet requerirá dos objetos **JTextField** que hayan registrado eventos de acción. Cuando se invoca a **actionPerformed**, el parámetro **ActionEvent** tiene al método **getSource()** para determinar el componente GUI con el que el usuario interactuó. Su método **actionPerformed** debe contener una estructura **if/else** de la siguiente forma:

```
if ( e.getSource() == entrada1 ) {
    // procesa la interacción entrada1 aquí
}
else { // e.getSource() == entrada2
    // procesa la interacción entrada2 aquí
}
```

donde **entrada1** y **entrada2** son referencias de **JTextField**.

- 25.13** Se dice que un entero es *primo* si sólo es divisible entre 1 y entre él mismo. Por ejemplo, 2, 3, 5 y 7 son números primos, pero 4, 6, 8 y 9, no lo son.
- Escriba un método que determine si un número es primo.
 - Utilice este método en un applet que determine e imprima todos los números primos entre 1 y 10,000. ¿Cuántos de estos 10,000 números realmente tiene que evaluar, antes de estar seguro de que encontró a todos los primos? Despliegue el resultado en un **JTextArea** que tenga la funcionalidad de desplazamiento.
 - De entrada, usted podría pensar que $n/2$ es el límite superior de los números que tiene que evaluar para ver si un número es primo, pero sólo necesita ir hasta la raíz cuadrada de n . ¿Por qué? Rescriba el programa y ejecútelos en ambas formas. Calcule un estimado de la mejoría en el rendimiento.
- 25.14** Escriba un método que tome un valor entero y que devuelva el número con sus dígitos a la inversa. Por ejemplo, dado el número 7631, el método debe devolver 1367. Incorpore el método en un applet que lea un valor del usuario. Despliegue el resultado del método en la barra de estado.
- 25.15** El *máximo común divisor (MCD)* de dos enteros es el entero más grande que divide en partes iguales a cada uno de los dos números. Escriba un método **mcd** que devuelva el máximo común divisor de dos enteros. Incorpore el método en un applet que lea dos valores del usuario. Despliegue el resultado del método en la barra de estado.
- 25.16** Escriba un método **puntosCalidad** que introduzca el promedio de un estudiante, y que devuelve 4 si el promedio de un estudiante es 90 a 100, 3 si el promedio es de 80 a 89, 2 si el promedio es de 70 a 79, 1 si el promedio es de 60 a 69 y 0 si el promedio es menor que 60. Incorpore el método en un applet que lea un valor del usuario. Despliegue el resultado del método en la barra de estado.
- 25.17** Escriba un applet que simule el lanzamiento de una moneda. Deje que el programa lance la moneda, cada vez que el usuario oprima el botón “**Lanzar**”. Cuente el número de veces que aparece cada lado de la moneda. Despliegue los resultados. El programa debe llamar a un método separado **tirar** que no tome argumentos y que devuelva **false** para la cruz y **true** para la cara. [Nota: Si el programa simula en forma realista el lanzamiento de la moneda, cada lado de la moneda debe aparecer aproximadamente en la mitad de las ocasiones que ésta se lance.]
- 25.18** Las computadoras tienen un papel cada vez más importante en la educación. Escriba un programa que ayude a un estudiante de primaria a aprender a multiplicar. Utilice **Math.random** para producir dos enteros positivos de un dígito. El programa debe desplegar después una pregunta en la barra de estado como

¿cuánto es 6 por 7?

El estudiante después debe escribir la respuesta en un **JTextField**. Su programa verifica la respuesta del estudiante. Si es correcta, dibuje la cadena “**Muy bien!**” en el applet, después haga otra pregunta. Si la respuesta es incorrecta, dibuje la cadena “**No. Pruebe otra vez.**” en el applet, después espere a que el estudiante intente otra vez repetidamente hasta que finalmente dé la respuesta correcta. Debe utilizar un método separado para generar cada nueva pregunta. Este método debe ser llamado una vez que el applet comience su ejecución, y cada vez que el usuario responda correctamente. Todos los dibujos del applet deben realizarse por medio del método **paint**.

- 25.19** Escriba un applet que juegue a “adivinar el número” de la siguiente manera: su programa elige el número a adivinar, seleccionando un entero al azar en el rango 1 a 1000. El applet despliega la indicación **Adivine un número entre 1 y 1000** junto a un **JTextField**. El jugador escribe un primer intento en el **JTextField** y opriime la tecla **Entrar**. Si el jugador no adivinó, su programa debe desplegar **Demasiado alto. Intente otra vez, o Demasiado bajo. Intente otra vez** en la barra de estado, para ayudar al jugador a “concentrarse” en la respuesta correcta, y debe limpiar el **JTextField** para que el usuario pueda introducir el siguiente intento.

Cuando el usuario introduzca la respuesta correcta, despliegue **Felicidades. Adivino el numero!** en la barra de estado, y limpíe el **JTextField** para que el usuario pueda jugar de nuevo. [Nota: La técnica para adivinar que empleamos en este problema es parecida a la de la *búsqueda binaria*.]

- 25.20** El máximo común divisor de los enteros **x** y **y** es el entero más grande que divide en partes iguales tanto a **x** como a **y**. Escriba un método recursivo **mcd** que devuelva el máximo común divisor de **x** y **y**. El **mcd** de **x** y **y** se define recursivamente de la siguiente forma: si **y** es igual que 0, entonces el **mcd(x, y)** es **x**; de lo contrario, **mcd(x, y)** es **mcd(y, x%y)**, donde % es el operador módulo. Utilice este método para reemplazar el que escribió en el applet del ejercicio 25.15.
- 25.21** Modifique el programa de craps de la figura 25.13 para permitir las apuestas. Inicialice en 1000 dólares la variable **saldoBanco**. Indique al usuario que introduzca una **apuesta**. Verifique que la **apuesta** sea menor o igual que **saldoBanco**, y si no es así, haga que el usuario reintroduzca una apuesta, hasta que introduzca una válida. Despues de que introduzca una **apuesta** correcta, ejecute un juego de craps. Si el jugador gana, incremente **saldoBanco** en el monto de la **apuesta** e imprima el nuevo **saldoBanco**. Si el jugador pierde, disminuya **saldoBanco** en el monto de la **apuesta**, imprima el nuevo **saldoBanco**, verifique si éste se ha vuelto cero, y si es así, imprima el mensaje **"Lo siento. Se quedó sin un centavo!"** Conforme progrese el juego, imprima varios mensajes para generar cierto "cotorreo", como **"Oh, va directo a la quiebra"**, o **"Ande, atrévase!"**, o **"Está ganando. Ahora es el momento de capitalizar!"**. Implemente el "cotorreo" como un método separado que elija al azar la cadena a desplegar.
- 25.22** Escriba un programa para simular el tiro de dos dados. El programa debe utilizar **Math.random** para tirar el primer dado, y debe utilizar **Math.random** nuevamente para tirar el segundo dado. La suma de los dos valores debe entonces calcularse. [Nota: Debido a que cada dado puede mostrar un valor entero entre 1 y 6, la suma de los valores variará de 2 a 12, en donde 7 es la suma más frecuente, y 2 y 12 son las sumas menos frecuentes. La figura 25.24 muestra las 36 posibles combinaciones de los dos dados. Su programa debe tirar el dado 36,000 veces. Utilice un arreglo con un solo subíndice para que lleve la cuenta del número de veces que cada posible suma aparece. Imprima los resultados en un formato tabular. Además, determine si los totales son razonables (es decir, existen seis formas de tirar un 7, por lo que aproximadamente un sexto de todos los tiros debe resultar en 7).]

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

Figura 25.24 Las 36 posibles salidas del tiro de dos dados.

Programación orientada a objetos con Java

Objetivos

- Comprender el encapsulamiento y el ocultamiento de información.
- Comprender los fundamentos de la abstracción de datos y los tipos de datos abstractos (ADTs).
- Crear ADTs en Java, a saber, clases.
- Crear, utilizar y destruir objetos.
- Controlar el acceso a las variables de instancia de objetos y a los métodos.
- Apreciar el valor de la orientación a objetos.
- Comprender el uso de la referencia **this**.
- Comprender las variables de clase y los métodos de clase.



Mi objetivo completamente sublime

Deberé lograrlo a tiempo.

W. S. Gilbert

¿Es éste un mundo en el que se deben esconder las virtudes?

William Shakespeare

Tus sirvientes públicos te sirven bien.

Adlai Stevenson

Pero acaso, para servir a nuestros fines personales,

¿Perdonamos el engaño a nuestros amigos?

Charles Churchill

Por sobre todas las cosas: sé auténtico.

William Shakespeare

No tengas amigos diferentes a ti mismo.

Confucio

Plan general

- 26.1 Introducción**
- 26.2 Implementación del tipo de dato abstracto Hora con una clase**
- 26.3 Alcance de una clase**
- 26.4 Creación de paquetes**
- 26.5 Inicialización de los objetos de una clase: Constructores**
- 26.6 Uso de los métodos *obtener* y *establecer***
- 26.7 Uso de la referencia *this***
- 26.8 Finalizadores**
- 26.9 Miembros estáticos de una clase**

Resumen • *Terminología* • *Errores comunes de programación* • *Buenas prácticas de programación* • *Tips de rendimiento* • *Observaciones de ingeniería de software* • *Ejercicios de autoevaluación* • *Respuestas a los ejercicios de autoevaluación* • *Ejercicios*

26.1 Introducción

Ahora estudiaremos la orientación a objeto en Java. Si usted ya leyó la introducción a la orientación a objetos en C++ (capítulo 16) podría saltar directo a la sección 26.2, en donde echamos un vistazo por primera vez a una implementación orientada a objetos en Java.

Revisemos brevemente algunos conceptos clave y la terminología de la orientación a objetos. La programación orientada a objetos (POO) *encapsula* datos (*atributos*) y métodos (*comportamientos*) dentro de *objetos*; los datos y los métodos de un objeto se encuentran íntimamente ligados entre sí. Los objetos tienen la propiedad de *ocultar la información*. Esto significa que aunque los objetos pueden saber cómo comunicarse entre sí, a través de *interfaces* bien definidas, por lo general a los objetos no se les permite saber cómo se implementan otros objetos; los detalles de implementación están ocultos dentro de los mismos objetos. Con toda seguridad es posible conducir un automóvil de manera efectiva sin conocer los detalles de cómo funcionan internamente los sistemas del motor, la transmisión y el escape. Veremos por qué el ocultamiento de información es tan importante para la buena ingeniería de software.

En C y en otros *lenguajes de programación por procedimientos*, la programación tiende a ser *orientada a acciones*. En Java, la programación es *orientada a objetos*. En C, la unidad de programación es la *función* (las cuales se conocen como *métodos* en Java). En Java, la unidad de programación es la *clase*, a partir de la cual se generan las *instancias* de todos los objetos (es decir, se crean). Las funciones no desaparecen en Java; en lugar de eso se encapsulan como métodos con los datos que procesan dentro de las “paredes” de las clases.

Los programadores en C se concentran en escribir funciones. Los conjuntos de acciones que realizan alguna tarea se agrupan en funciones, y las funciones se agrupan para formar programas. Los datos son importantes en C, pero la idea es que los datos existen primordialmente para apoyar las acciones que realizan las funciones. En la especificación de un sistema, los *verbos* ayudan al programador en C a determinar el conjunto de funciones que trabajarán juntas para implementar el sistema.

Los programadores en Java se concentran en crear sus propios *tipos definidos por el usuario* llamados *clases*. A las clases también se les denomina *tipos definidos por el programador*. Toda clase contiene datos, así como el conjunto de métodos que manipulan estos datos. A los datos que componen una clase se les llama *variables de instancia* (o *datos miembro*, en C++). Así como a una instancia de un tipo de dato predefinido como `int` se le llama *variable*, a una instancia de un tipo de dato definido por el usuario (es decir, a una clase) se le llama *objeto*. El foco de atención en Java se centra en los objetos, en lugar de en los métodos. Los *sustantivos* que se encuentran en las especificaciones de un sistema ayudan al programador en Java a determinar el conjunto de clases que utilizará para comenzar el proceso de diseño. Después, se utilizan las clases para crear las instancias de los objetos que trabajarán juntas para implementar un sistema.

Este capítulo explica cómo crear objetos, un tema al que nos gusta llamar *programación basada en objetos* (PBO). En el capítulo 27 introducimos la *herencia* y el polimorfismo, dos tecnologías clave que permiten

la verdadera *programación orientada a objetos (POO)*. Aunque no explicaremos con detalle la herencia hasta el capítulo 27, ésta es parte de toda definición de una clase en Java.



Tip de rendimiento 26.1

Todos los objetos en Java se pasan por referencia. Sólo se pasa la dirección de memoria, no una copia de todo el objeto (como se haría en un paso por valor).



Observación de ingeniería de software 26.1

Es importante escribir programas que sean claros y fáciles de mantener. La regla es el cambio, en lugar de la excepción. Los programadores deben prever que su código será modificado. Como veremos pronto, las clases facilitan la modificación de un programa.

26.2 Implementación del tipo de dato abstracto Hora con una clase

La aplicación de la figura 26.1 consta de dos clases, **Horal** y **PruebaHora**. La clase **Horal** se define en el archivo **Horal.java** (especificado en la línea de comentario 1) y la clase **PruebaHora** se define en el archivo **PruebaHora.java** (especificada en la línea de comentario 49). [Nota: Todos los programas de este libro que contienen más de un archivo, comienzan con un comentario que indica el número de la figura y el nombre del archivo.] Aunque estas dos clases están definidas en archivos separados, numeramos consecutivamente las líneas del programa a lo largo de ambos archivos, por motivos de explicación en el texto. Es importante observar que estas clases *deben* definirse en archivos separados.



Observación de ingeniería de software 26.2

*Las definiciones de las clases que comienzan con la palabra reservada **public** deben almacenarse en un archivo que tiene el mismo nombre que la clase, y terminar con la extensión de archivo **.java**.*



Error común de programación 26.1

Definir más de una clase pública en el mismo archivo, es un error de sintaxis.

```

1 // Figura 26.1: Horal.java
2 // Definición de la clase Horal
3 import java.text.DecimalFormat; // se utiliza para dar formato al número
4
5 // Esta clase mantiene la hora en formato de 24 horas
6 public class Horal extends Object {
7     private int hora; // 0 - 23
8     private int minuto; // 0 - 59
9     private int segundo; // 0 - 59
10
11    // El constructor Horal inicializa en cero cada variable
12    // de instancia. Garantiza que cada vez que inicia el objeto Horal
13    // lo hace en un estado consistente.
14    public Horal()
15    {
16        estableceHora( 0, 0, 0 );
17    } // fin del constructor Horal
18
19    // Establece un nuevo valor de hora por medio del horario universal.
20    // Realiza validaciones a los datos. Establece en cero a los valores
21    // inválidos.
22    public void estableceHora( int h, int m, int s )
23    {
24        hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
25        minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );

```

Figura 26.1 Implementación del tipo de dato abstracto **Horal** como una clase; **Horal.java**.
(Parte 1 de 2.)

```

25     segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
26 } // fin del método estableceHora
27
28 // Convierte a String en formato de horario universal
29 public String aCadenaUniversal()
30 {
31     DecimalFormat dosDigitos = new DecimalFormat( "00" );
32
33     return dosDigitos.format( hora ) + ":" +
34         dosDigitos.format( minuto ) + ":" +
35         dosDigitos.format( segundo );
36 } // fin del método aCadenaUniversal
37
38 // Convierte a String en formato de horario estándar
39 public String toString()
40 {
41     DecimalFormat dosDigitos = new DecimalFormat( "00" );
42
43     return ( ( hora == 12 || hora == 0 ) ? 12 : hora % 12 ) +
44         ":" + dosDigitos.format( minuto ) +
45         ":" + dosDigitos.format( segundo ) +
46         ( hora < 12 ? " AM" : " PM" );
47 } // fin del método toString
48 } // fin de la clase Horal

```

Figura 26.1 Implementación del tipo de dato abstracto **Horal1** como una clase; **Horal1.java**. (Parte 2 de 2.)

```

49 // Figura 26.1: PruebaHora.java
50 // Clase PruebaHora para ejercitarse la clase Horal
51 import javax.swing.JOptionPane;
52
53 public class PruebaHora {
54     public static void main( String args[] )
55     {
56         Horal h = new Horal(); // llama al constructor Horal
57         String salida;
58
59         salida = "La hora universal inicial es: " +
60             h.aCadenaUniversal() +
61             "\nLa hora estandar inicial es: " +
62             h.toString() +
63             "\nLlamada implicita a toString(): " + h;
64
65         h.estableceHora( 13, 27, 6 );
66         salida += "\n\nLa hora universal despues de estableceHora es: " +
67             h.aCadenaUniversal() +
68             "\nLa hora estandar despues de estableceHora es: " +
69             h.toString();
70
71         h.estableceHora( 99, 99, 99 ); // todos son valores inválidos
72         salida += "\n\nDespues de intentar establecer valores invalidos: " +
73             "\nHora universal: " + h.aCadenaUniversal() +

```

Figura 26.1 Implementación del tipo de dato abstracto **Horal1** como una clase; **PruebaHora.java**. (Parte 1 de 2.)

```

74         "\nHora estandar: " + h.toString());
75
76     JOptionPane.showMessageDialog( null, salida,
77         "Probando la clase Horal",
78         JOptionPane.INFORMATION_MESSAGE );
79
80     System.exit( 0 );
81 } // fin de main
82 } // fin de la clase PruebaHora

```

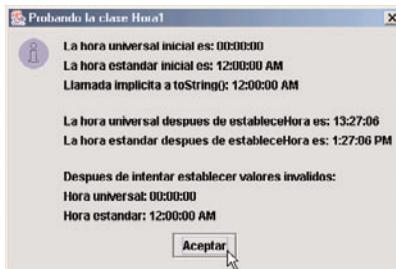


Figura 26.1 Implementación del tipo de dato abstracto **Horal1** como una clase; **PruebaHora.java**. (Parte 2 de 2.)

La figura 26.1 (líneas 1 a 48) contienen una sencilla definición para la clase **Horal1**. Nuestra definición de la clase **Horal1** comienza en la línea 6

```
public class Horal1 extends Object {
```

la cual indica que la clase **Horal1 extiende** a la clase **Object** (del paquete **java.lang**). Recuerde que usted realmente no crea una definición de clase “desde cero”. De hecho, cuando crea una definición de clase, siempre utiliza piezas de definiciones de clase existentes. Java utiliza la *herencia* para crear nuevas clases a partir de definiciones de clases existentes. La palabra reservada **extends** seguida por el nombre de clase **Object** indica la clase (en este caso **Horal1**) a partir de la cual nuestras nuevas clases heredan sus piezas existentes. En esta relación de herencia, a **Object** se le llama *superclase* o *clase base* y a **Horal1** se le llama *subclase* o *clase derivada*. El uso de la herencia da como resultado una nueva definición de clase que tiene *atributos* (datos) y *comportamientos* (métodos) de la clase **Object**, así como las nuevas características que agregamos a nuestra definición de la clase **Horal1**. Toda clase en Java es una subclase de **Object**. Por lo tanto, cada clase hereda los once métodos definidos por la clase **Object**. Un método clave de **Object** es **toString**, el cual explicaremos más adelante en esta sección. Explicaremos otros métodos de la clase **Object** a través del libro, cuando sea necesario.

Observación de ingeniería de software 26.3



Toda clase definida en Java debe ser una extensión de otra clase. Si la clase no utiliza explícitamente la palabra reservada **extends** en su definición, esta clase implícitamente se extiende de **Object**.

El *cuerpo* de la definición de una clase se delinea con una llave izquierda y una llave derecha (**{ y }**) en las líneas 6 a 48. La clase **Horal1** contiene tres variables de instancia enteras, **hora**, **minuto** y **segundo**, que representan el tiempo en formato de *horario universal* (*formato de reloj de 24 horas*).

Las palabras reservadas **public** y **private** son *modificadores de acceso a miembros*. Las variables de instancia o métodos que se declaran con el modificador de acceso a miembros **public** son accesibles desde cualquier punto en donde el programa haga referencia al objeto **Horal1**. Las variables de instancia o métodos que se declaran con el modificador de acceso a miembros **private** solamente están accesibles para los métodos de la clase. A cada variable de instancia o definición método le debe anteceder un modificador de acceso a miembros. Los modificadores de acceso a miembros pueden aparecer varias veces en cualquier orden en una definición de clase.

Buena práctica de programación 26.1



Agrupe los miembros de acuerdo con los modificadores de acceso a miembros dentro de la definición de una clase, para mayor claridad y legibilidad.

Error común de programación 26.2



El hecho de que un método que no es un miembro de una clase en particular intente acceder a un miembro privado de dicha clase, es un error de sintaxis.

Las tres variables de instancia enteras **hora**, **minuto** y **segundo** se declaran (líneas 7 a 9) con el modificador de acceso a miembros **private**. Esto indica que las variables de instancia de la clase son las únicas accesibles para los métodos de la clase. Cuando se crea la instancia de un objeto de la clase, dichas variables de instancia se encapsulan dentro del objeto y se puede acceder a ellas solamente a través de los métodos del objeto de la clase (por lo general, a través de los métodos públicos de la clase). Las variables de instancia normalmente se declaran como **private**, y los métodos por lo general se declaran como **public**. Es posible tener métodos privados y datos públicos, como veremos más adelante. A los métodos privados a menudo se les llama *métodos de utilidad* o *métodos de ayuda* debido a que solamente se les puede llamar mediante otros métodos de la clase, y se utilizan para soportar la operación de dichos métodos. El uso de datos públicos no es común y es una práctica peligrosa de programación.

Buena práctica de programación 26.2



*Nosotros preferimos listar primero a las variables de instancia **private** de una clase, para que conforme lea el código, vea los nombres y los tipos de dichas variables, antes de utilizarlas en los métodos de la clase.*

Buena práctica de programación 26.3



A pesar del hecho de que los miembros públicos y privados pueden repetirse y mezclarse, primero liste en un grupo a todos los miembros privados de la clase, y después liste en otro grupo a todos los miembros públicos.

Observación de ingeniería de software 26.4



Mantenga privadas todas las variables de instancia. Cuando sea necesario, proporcione métodos públicos para establecer los valores de variables de instancia privadas y para obtener los valores de variables de instancia privadas. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual reduce los errores y mejora la posibilidad de modificación del programa.

Observación de ingeniería de software 26.5



Los métodos tienden a caer en diversas categorías: métodos que obtienen los valores a partir de variables de instancia privadas; métodos que establecen los valores de las variables de instancia privadas; métodos que implementan los servicios de la clase; y métodos que realizan distintos mecanismos para la clase, tales como la inicialización de los objetos de las clases, la asignación de los objetos de las clases, y la conversión entre clases y los tipos predefinidos, o entre clases y otras clases.

Los métodos de acceso pueden leer o desplegar datos. Otro uso común para los métodos de acceso es probar la verdad o falsedad de condiciones, por lo general, a dichos métodos se les llama *métodos predicados*. Un ejemplo de un método predicado podría ser el método **estaVacia** para cualquier clase contenedora; una clase capaz de almacenar muchos objetos, tales como una lista ligada, una pila o una cola. Un programa podría probar **estaVacia** antes de intentar leer otro elemento del objeto contenedor. Un programa podría probar **estaLlena** antes de intentar insertar otro elemento en el objeto contenedor.

La clase **Horal** contiene los siguientes métodos públicos, **Horal** (línea 14), **estableceHora** (línea 21), **aCadenaUniversal** (línea 29) y **toString** (línea 39). Éstos son *métodos públicos, servicios públicos* o la *interfaz de la clase*. Estos métodos los utilizan los *clientes* (es decir, porciones de un programa que son usuarios de una clase) de las clases para manipular los datos almacenados en los objetos de la clase.

Los clientes de una clase utilizan referencias para interactuar con objetos de la clase. Por ejemplo, el método **paint** dentro de un applet es un cliente de la clase **Graphics**; **paint** utiliza una referencia al objeto **Graphics** (tal como **g**), la cual lo recibe como argumento para dibujar en el applet, por medio de la llamada a los métodos que son servicios públicos de la clase **Graphics** (tales como **drawString**, **drawLine**, **drawOval** y **drawRect**).

Observe el método con el mismo nombre que la clase (línea 14); se trata del método *constructor* de la clase. Un constructor es un método especial que inicializa las variables de instancia del objeto de la clase. Se llama a un método constructor de la clase, cada vez que se crea la instancia de un objeto de dicha clase. Este constructor simplemente llama al método **estableceHora** de la clase (la cual explicaremos pronto) con los valores de la hora, el minuto y el segundo especificados como 0.

Los constructores pueden tomar argumentos pero no pueden devolver valor alguno. Una diferencia importante entre los constructores y otros métodos es que a los constructores *no se les permite especificar un tipo de dato de retorno* (incluso **void**). Por lo general, los constructores son métodos públicos de una clase. Más adelante explicaremos los métodos no públicos.

Error común de programación 26.3



Intentar declarar un tipo de retorno para un constructor y/o intentar devolver un valor desde un constructor; es un error lógico. Java permite a otros métodos de la clase tener el mismo nombre de la clase y especificar los tipos de retorno. Dichos métodos no son constructores y no se les llamará cuando se genere la instancia de un objeto de la clase.

El método **estableceHora** (línea 21) es un método público que recibe tres argumentos enteros y los utiliza para establecer la hora. Cada argumento se prueba dentro de una expresión condicional que determina si el valor se encuentra en rango. Por ejemplo, el valor **hora** debe ser mayor que 0 igual que 0 y menor que 24, debido a que representamos el tiempo con formato de tiempo universal (0-23 para la hora, 0-59 para el minuto y 0-59 para el segundo). Cualquier valor fuera de este rango es un valor inválido y se establece en cero, lo que asegura que el objeto **Horal** siempre contiene un dato válido. A esto se le llama *mantener al objeto en estado consistente*. En casos en los que se proporcionan datos inválidos para **estableceHora**, el programa podría querer indicar que se intentó establecer un valor inválido. Exploraremos esta posibilidad en los ejercicios.

Buena práctica de programación 26.4



Siempre defina una clase de manera que sus variables de instancia se mantengan en un estado consistente.

El método **aCadenaUniversal** (línea 29) no toma argumentos y devuelve un **String**. Este método produce una cadena con la hora en formato universal que consta de seis dígitos, dos para la hora, dos para el minuto y dos para el segundo. Por ejemplo, 13:30:07 representa 1:30:07 PM. La línea 31 crea una instancia de la clase **DecimalFormat** (del paquete **java.text** importado en la línea 3) para ayudar a mantener la hora en formato universal. El objeto **dosDigitos** se inicializa con la *cadena de control de formato "00"*, la cual indica que el formato del número debe consistir en dos dígitos, cada 0 es una posición para un dígito. Si el número al que se le da formato es de un solo dígito, a éste le antecede un 0 (es decir, a 8 se le da formato como 08). La instrucción **return** de las líneas 33 a 35 utilizan el método **format** (que devuelve un **String** con formato, el cual contiene el número) del objeto **dosDigitos** para dar formato a los valores de la hora, el minuto y el segundo como cadenas de dos dígitos. Dichas cadenas se concatenan con el operador + (separado por punto y coma), y devuelto desde el método **aCadenaUniversal**.

El método **toString** (línea 29) no toma argumentos y devuelve un **String**. Este método produce una cadena con formato de hora estándar que consta de los valores **hora**, **minuto** y **segundo** separados por dos puntos y un indicador AM o PM, como en **1:27:06 PM**. Este método utiliza las mismas técnicas de **DecimalFormat** que el método **aCadenaUniversal**, para garantizar que los valores para **minuto** y **segundo** aparezcan con dos dígitos. El método **toString** es especial, debido a que heredamos un método **toString** de la clase **Object** con exactamente la primera línea que nuestro **toString** de la línea 39. El método **toString** original de la clase **Object** es una versión general que utilizamos con frecuencia como un contenedor que puede redefinirse mediante una subclase (similar a los métodos **init**, **start** y **paint** de la clase **JApplet**). Nuestra versión reemplaza a la versión que heredemos para proporcionar un método **toString** más apropiado para nuestra clase. A esto se le conoce como *redefinir* la definición original del método (explicada con detalle en el capítulo 27).

Una vez que se define la clase, ésta puede utilizarse como un tipo en una declaración como

```
Horal atardecer,      // referencia al objeto de tipo Horal
arregloHora[];       // referencia al arreglo de objetos de Horal
```

El nombre de la clase es un nuevo especificador de tipo. Existen muchos objetos de una clase, así como pueden existir muchas variables de tipos de datos primitivos tales como **int**. El programador puede crear tantos

nuevos tipos de clases como sea necesario; ésta es una de las razones por las cuales a Java se le conoce como un *lenguaje extensible*.

La aplicación de la figura 26.1 (líneas 48 a 82) utiliza la clase **Horal1**. El método **main** de la clase **PruebaHora** declara e inicializa una instancia de la clase **Horal1** llamada **h** con la línea 56

```
Horal1 h = new Horal1(); // llama al constructor Horal1
```

Cuando se crea la instancia del objeto, el operador **new** asigna la memoria en la que se almacenará el objeto de **Horal1**, después **new** llama al constructor **Horal1** para inicializar las variables de instancia del nuevo objeto de **Horal1**. El constructor invoca al método **estableceHora** para inicializar explícitamente cada variable de instancia privada en 0. El operador **new** devuelve entonces una referencia al nuevo objeto, y dicha referencia se asigna a **h**. De manera similar, la línea 31 de la clase **Horal1** utiliza **new** para asignar la memoria para el objeto **DecimalFormat**, y luego llama al constructor **DecimalFormat** con el argumento “00” para indicar la cadena de control de formato del número.

Observación de ingeniería de software 26.6



Cada vez que new crea un objeto de la clase, se llama al constructor de dicha clase para inicializar las variables de instancia del nuevo objeto.

Observe que la clase **Horal1** no se importó hacia archivo **PruebaHora.java**. En realidad, cada clase en java es parte de un *paquete* (como las clases del API de JAVA). Si el programador no especifica el paquete para una clase, la clase se coloca automáticamente en el *paquete predeterminado*, el cual incluye las clases compiladas en el directorio actual. Si una clase se encuentra en el mismo paquete que el de la clase que la utiliza, no se requiere una instrucción **import**. Importamos clases desde el API de Java debido a que sus archivos **.class** no se encuentran en el mismo paquete con cada programa que escribimos. En la sección 26.4 explicamos cómo definir sus propios paquetes de clases.

La línea 57 declara una referencia a un **String** llamada **salida** que almacenará la cadena que contiene los resultados a desplegarse en el diálogo de mensaje. Las líneas 59 a 63 agregan la hora en formato universal a **salida** (al enviar un mensaje **aCadenaUniversal**, hacia el objeto al que hace referencia **h**) y en formato de tiempo estándar (al enviar el mensaje **toString**, hacia el objeto al que hace referencia **h**) para confirmar que los datos se inicializaron apropiadamente. Observe la línea 63

```
"\nLlamada implícita a toString(): " + h;
```

En Java, el operador **+** puede utilizarse para concatenar cadenas. Aplicar el operador **+** a una cadena y a un objeto da como resultado una llamada implícita al método **toString** del objeto, el cual convierte al objeto en una cadena. El operador **+** después concatena las dos cadenas para producir una sola. Las líneas 62 y 63 muestran que usted puede llamar a **toString** tanto implícitamente como explícitamente, en una operación de concatenación de cadenas.

La línea 65

```
h.estableceHora( 13, 27, 6 );
```

envía el mensaje **estableceHora** al objeto al cual **h** hace referencia, para modificar nuevamente la hora de **salida** en ambos formatos y confirmar que la hora se estableció correctamente.

Para mostrar que el método **estableceHora** valida los valores que se le pasan, la línea 71

```
h.estableceHora( 99, 99, 99 ); // todos son valores inválidos
```

llama al método **estableceHora** e intenta establecer las variables de instancia con valores válidos. Luego, las líneas 72 a 74 agregan nuevamente la hora a **salida** en ambos formatos para confirmar que **estableceHora** valida los datos. Las líneas 76 a 78 despliegan un cuadro de mensaje con los resultados de nuestro programa. Observe en las dos últimas líneas de la ventana de salida que la hora se establece en medianoche; el valor predeterminado del objeto **Horal1**.

Ahora que hemos visto nuestra primera clase que no es un applet ni una aplicación, consideremos varios puntos del diseño de clases.

De nuevo, observe que las variables de instancia **hora**, **minuto** y **segundo** se declaran en donde se definen. Aquí, la filosofía es que la representación de los datos reales utilizada dentro de las clases no es asunto de los clientes de la clase. Por ejemplo, sería perfectamente razonable para la clase representar la hora interna-

mente como el número de segundos desde medianoche. Los clientes podrían utilizar los mismos métodos públicos y obtener los mismos resultados sin darse cuenta de esto. En este sentido, se dice que la implementación de una clase está *oculta* a sus clientes. El ejercicio 26.10 le pide que haga las modificaciones precisas a la clase **Hora1** de la figura 26.1 para mostrar que no existe un cambio visible para los clientes de la clase.

Observación de ingeniería de software 26.7



El ocultamiento de información promueve la capacidad de modificación del programa y simplifica la percepción de los clientes respecto a la clase.



Observación de ingeniería de software 26.8

Los clientes de una clase pueden (y deben) utilizar la clase sin conocer los detalles de implementación de la clase. Si cambia la implementación de la clase (por ejemplo, para mejorar el rendimiento), la interfaz proporcionada permanece constante, el código fuente de los clientes de la clase no necesitan modificación. Esto hace mucho más fácil la modificación de los sistemas.

En este programa, el constructor **Hora1** simplemente inicializa las variables de instancia en 0 (es decir, el equivalente militar de las 12 AM). Esto asegura que el objeto se crea con un *estado consistente* (es decir, todos los valores de las variables de instancia son válidos). Los valores no válidos no pueden almacenarse en las variables de instancia del objeto **Hora1** debido a que el constructor se llama cuando se crea el objeto **Hora1**, y los intentos subsiguientes de un cliente por modificar las variables de instancia se examinan mediante el método **estableceHora**.

Las variables de instancia pueden inicializarse cuando se declaran en el cuerpo de la clase, por medio del constructor de la clase, o se les puede asignar valores por medio de instrucciones *establecer*. Las variables de instancia que el programador no inicializa explícitamente, las inicializa el compilador (las variables numéricas primitivas se establecen en 0, las booleanas en **false** y las referencias se establecen en **null**).



Buena práctica de programación 26.5

Inicialice las variables de instancia de una clase en el constructor de esa clase.

Es interesante que los métodos **aCadenaUniversal** y **toString** no tomen argumentos. Esto se debe a que estos métodos saben implícitamente que van a manipular las variables de instancia del objeto **Hora1** particular para el que se invocaron. Esto hace las llamadas a los métodos más concisas que las llamadas convencionales a funciones en la programación por procedimientos. También reduce la probabilidad de pasar los argumentos incorrectos, los tipos incorrectos de los argumentos y/o el número incorrecto de argumentos, como sucede con frecuencia en las llamadas a funciones en C.



Observación de ingeniería de software 26.9

Con frecuencia, utilizar un método de programación orientada a objetos simplifica las llamadas a los métodos, al reducir el número de parámetros a pasar. Este beneficio de la programación orientada a objetos se deriva del hecho de que el encapsulamiento de las variables de instancia y de los métodos dentro de un objeto le da a los métodos el derecho de acceso a las variables de instancia.

Las clases simplifican la programación debido a que el cliente (o usuario del objeto de la clase) solamente necesitan preocuparse por las operaciones públicas encapsuladas en el objeto. Por lo general, dichas operaciones están diseñadas para que estén orientadas al cliente en lugar de a la implementación. Los clientes no necesitan preocuparse por la implementación de la clase. La interfaz cambia, pero con menos frecuencia que las implementaciones. Cuando la implementación cambia, el código que depende de la implementación debe cambiar en concordancia. Al ocultar la implementación eliminamos la posibilidad de que otras partes del programa se hagan dependientes de los detalles de la implementación de la clase.

Con frecuencia, las clases no tienen que crearse “desde cero”. En lugar de eso, pueden *derivarse* desde otras clases que proporcionan operaciones que las nuevas clases pueden utilizar, o las clases pueden incluir como miembros objetos de otras clases. Tal *reutilización de software* puede mejorar enormemente la productividad del programador. A la derivación de clases a partir de clases existentes se le llama *herencia* y la explicaremos con detalle en el capítulo 27. A la inclusión de objetos de clases como miembros de otras clases se le llama *composición* o *agregación*, y la explicaremos más adelante en este capítulo.

26.3 Alcance de una clase

Las variables de instancia y los métodos de una clase pertenecen al *alcance de dicha clase*. Dentro del alcance de dicha clase, los miembros están accesibles de inmediato para todos los métodos de la clase y se puede hacer referencia a ellos simplemente por su nombre. Fuera del alcance de la clase, no se puede hacer referencia a los miembros de la clase directamente por su nombre. Sólo se puede acceder a dichos miembros de la clase (tales como miembros públicos) que son visibles por medio de un “manipulador” (es decir, se puede hacer referencia a los miembros con un tipo de dato primitivo por medio de **nombreReferenciaObjeto.nombreVariablePrimitiva**, y se puede hacer referencia a los miembros del objeto por medio de **nombreReferenciaObjeto.nombreMiembroObjeto**).

Las variables definidas en un método sólo se conocen en dicho método (es decir, son variables locales a dicho método). Se dice que dichas variables tienen alcance de bloque. Si un método define una variable con el mismo nombre que la variable con alcance de clase (es decir, una variable de instancia), la variable con alcance de clase se oculta en la variable local con alcance de método. Se puede acceder a una variable de instancia oculta en un método, al anteceder a su nombre la palabra reservada **this** y el operador punto, como en **this.x**. Más adelante en este capítulo, explicaremos la palabra reservada **this**.

26.4 Creación de paquetes

Como hemos visto en casi cada ejemplo del libro, las clases y las *interfaces* (que explicaremos en el capítulo 27) de las bibliotecas existentes, tales como la API de Java, pueden importarse dentro de un programa en Java. Cada clase e interfaz del API de Java pertenece a un paquete específico que contiene un grupo de clases e interfaces relacionadas. En realidad, los paquetes son estructuras de directorios que se utilizan para organizar las clases y las interfaces. Los paquetes proporcionan un mecanismo para la *reutilización de software*. Una de las metas de los programadores es crear componentes reutilizables de software, de manera que no sea necesario redefinir el código repetidamente en cada programa. Otro beneficio de los paquetes es que proporcionan una convención para los *nombres de clase únicos*. Con cientos de miles de programas en Java alrededor del mundo, existen muchas posibilidades de que los nombres que usted elija para las clases tengan conflicto con los nombres que otros programadores utilizan para sus clases.

La aplicación de la figura 26.2 ilustra la manera de crear su propio paquete y cómo utilizar una clase a partir de dicho paquete dentro de un programa.

```
1 // Figura 26.2: Horal.java
2 // Definición de la clase Horal
3 package com.deitel.chtp4.Cap26;
4 import java.text.DecimalFormat; // utilizado para dar formato al número
5
6 // Esta clase mantiene la hora en formato de 24 horas
7 public class Horal extends Object {
8     private int hora; // 0 - 23
9     private int minuto; // 0 - 59
10    private int segundo; // 0 - 59
11
12    // El constructor Horal inicializa en cero cada variable
13    // de instancia. Garantiza que cada objeto Horal inicia en un
14    // estado consistente.
15    public Horal()
16    {
17        estableceHora( 0, 0, 0 );
18    } // fin del constructor Horal
19
20    // Establece un nuevo valor de hora utilizando la hora militar. Realiza
```

Figura 26.2 Creación de un paquete para reutilización de software; **Horal.java**. (Parte 1 de 2.)

```

21 // validaciones de datos. Establece en cero a los
22 // valores inválidos.
23 public void estableceHora( int h, int m, int s )
24 {
25     hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
26     minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
27     segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
28 } // fin del método estableceHora
29
30 // Convierte a String en hora de formato universal
31 public String aCadenaUniversal()
32 {
33     DecimalFormat dosDigitos = new DecimalFormat( "00" );
34
35     return dosDigitos.format( hora ) + ":" +
36         dosDigitos.format( minuto ) + ":" +
37         dosDigitos.format( segundo );
38 } // fin del método aCadenaUniversal
39
40 // Convierte a String en hora de formato estándar
41 public String toString()
42 {
43     DecimalFormat dosDigitos = new DecimalFormat( "00" );
44
45     return ( (hora == 12 || hora == 0) ? 12 : hora % 12 ) +
46         ":" + dosDigitos.format( minuto ) +
47         ":" + dosDigitos.format( segundo ) +
48         ( hora < 12 ? " AM" : " PM" );
49 } // fin del método toString
50 } // fin de la clase Horal

```

Figura 26.2 Creación de un paquete para reutilización de software; **Horal.java**. (Parte 2 de 2.)

```

51 // Figura 26.2: PruebaHora.java
52 // Clase PruebaHora para utilizar la clase importada Horal
53 import javax.swing.JOptionPane;
54 import com.deitel.chtp4.Cap26.Horal; // importa a la clase Horal
55
56 public class PruebaHora {
57     public static void main( String args[] )
58     {
59         Horal h = new Horal();
60
61         h.estableceHora( 13, 27, 06 );
62         String salida =
63             "La hora universal es: " + h.aCadenaUniversal() +
64             "\nLa hora estandar es: " + h.toString();
65
66         JOptionPane.showMessageDialog( null, salida,
67             "Empacando la clase Horal para reutilizarla",
68             JOptionPane.INFORMATION_MESSAGE );
69
70         System.exit( 0 );

```

Figura 26.2 Creación de un paquete para reutilización de software; **PruebaHora.java**. (Parte 1 de 2.)

```
71 } // fin de main
72 } // fin de la clase PruebaHora
```

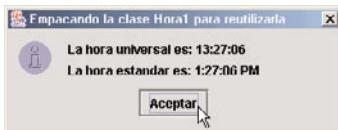


Figura 26.2 Creación de un paquete para reutilización de software; **PruebaHora.java**.
(Parte 2 de 2.)

Los pasos para crear clases reutilizables son:

1. Declare una clase pública. Si la clase no es pública, solamente puede ser utilizada por otras clases en el mismo paquete.
2. Elija un nombre de paquete, y agregue una *instrucción package* al archivo de código fuente para la declaración de la clase reutilizable. Solamente puede haber una instrucción **package** en el archivo de código fuente de Java, y debe anteceder a todas las demás declaraciones e instrucciones en el archivo.
3. Compile la clase de manera que se coloque en el lugar apropiado de la estructura de directorio del paquete.
4. Importe la clase reutilizable dentro de un programa, y utilice la clase.

Para el *paso 1*, elegimos utilizar la clase pública **Hora1** de la figura 26.1. No hicimos modificaciones a la implementación de la clase, de modo que no explicaremos nuevamente los detalles de implementación de dicha clase.

Para satisfacer el *paso 2*, agregamos una instrucción **package** al principio del archivo. La línea 3

```
package com.deitel.chtp4.Cap26;
```

utiliza la *instrucción package* para definir un paquete con el nombre **com.deitel.chtp4.Cap26**. Al colocar la instrucción **package** al principio del archivo de código fuente en Java indicamos que la clase definida en el archivo es parte del paquete especificado. Las únicas instrucciones en Java que aparecen fuera de las llaves de la definición de la clase son las instrucciones **package** e **import**.

Observación de ingeniería de software 26.10



Un archivo de código fuente en Java tiene el siguiente orden: una instrucción **package** (si existe alguna), instrucción **import** (si existen), y las definiciones de las clases. Solamente una de las definiciones de las clases puede ser pública. Las demás clases en el archivo también se colocan en el paquete, pero no son reutilizables. Éstas se encuentran en el paquete para soportar a la clase reutilizable del archivo.

En un esfuerzo por proporcionar un nombre único para cada paquete. Sun Microsystems especifica una convención para asignar nombres a los paquetes. Cada nombre de paquete debe comenzar con el nombre de su dominio de Internet en orden inverso. Por ejemplo, nuestro dominio de Internet es **deitel.com**, de modo que el nombre de nuestro paquete inicia como **com.deitel**. Si su nombre de dominio es **suescuela.edu** el nombre del paquete que usted utilizaría es **edu.suescuela**. Después de invertir el nombre de dominio, puede elegir cualquier nombre que desee para su paquete. Si usted forma parte de una empresa con muchas divisiones, o de una universidad con muchas escuelas, podría utilizar el nombre de su división o escuela como el siguiente nombre del paquete. Elegimos utilizar **chtp4** como el siguiente nombre de nuestro paquete para indicar que esta clase es parte del libro. El último nombre en nuestro paquete especifica que es para el capítulo 26 (**Cap26**). [Nota: Utilizaremos nuestros propios paquetes a lo largo del libro. Usted puede determinar el capítulo en el que nuestras clases reutilizables están definidas, observando el último nombre de la instrucción **import**.]

El *paso 3* consiste en compilar la clase para almacenarla en el paquete apropiado. Cuando se compila un archivo en Java que contiene una instrucción **package**, el archivo de clase que resulta se coloca en la estructura de directorio especificada por la instrucción **package**. La instrucción **package** de la figura 26.2 indica que la clase **Hora1** debe colocarse en el directorio **Cap26**. Los otros nombres, **com**, **deitel** y **chtp4**, tam-

bien son directorios. Los nombres de directorios en la instrucción **package** especifican la ubicación exacta de las clases en el paquete. Si estos directorios no existen antes de la compilación de la clase, el compilador los crea.

Cuando se compila una clase dentro de un paquete, la opción (**-d**) de la línea de comando provoca que el compilador **javac** genere los directorios apropiados, basándose en la instrucción **package** de la clase. Además, la opción especifica en dónde crear (o localizar) los directorios. Por ejemplo, en una ventana de comando, utilizamos el comando de compilación

```
javac -d . Horal.java
```

para especificar que el primer directorio de nuestro paquete debe colocarse en el directorio actual. El **.** después de **-d** del comando anterior representa el directorio actual en los sistemas operativos Windows, UNIX y Linux (y muchos otros también). Después de ejecutar el comando de compilación, el directorio actual contiene un directorio llamado **com**; **com** contiene un directorio llamado **deitel**; **deitel** contiene un directorio llamado **chtp4**, y **chtp4** contiene un directorio llamado **Cap26**. En el directorio **Cap26** puede encontrar el archivo **Horal.class**. [Nota: Si no utiliza la opción **-d**, entonces primero debe copiar o mover el archivo de la clase al directorio de paquete apropiado después de compilarlo.]

El nombre del paquete es parte del nombre de la clase. El nombre de la clase en este ejemplo es en realidad **com.deitel.chtp4.Cap26.Horal**. Usted puede utilizar este nombre *completo* en sus programas, o puede importar la clase y utilizarla con su nombre simple (**Horal**) en el programa. Si otro paquete también contiene una clase **Horal**, se puede utilizar el nombre completo de la clase para distinguir entre las clases y evitar un *conflicto de nombres* (también llamado *colisión de nombres*).

Una vez que la clase se compila y se almacena en el paquete, ésta puede importarse dentro de los programas (*Paso 4*). La línea 54

```
import com.deitel.chtp4.Cap26.Horal; // importa la clase Horal
```

especifica que la clase **Horal** debe importarse para utilizarla en la clase **PruebaHora**. [Nota: Las clases del paquete nunca necesitan importar otras clases del mismo paquete.]

26.5 Inicialización de los objetos de una clase: Constructores

Cuando se crea un objeto, sus miembros pueden inicializarse por medio de un método *constructor*. Un constructor es un método con el mismo nombre que la clase (con sensibilidad a mayúsculas y minúsculas). El programador proporciona el constructor que se invoca de manera automática cada vez que se crea la instancia de un objeto de la clase. Las variables de instancia pueden inicializarse implícitamente con sus valores predeterminados (0 para los tipos numéricos primitivos, **false** para los booleanos y **null** para las referencias), y pueden inicializarse en el constructor de la clase, o posteriormente a la creación del objeto. Los constructores no pueden especificar tipos de retorno o valores de retorno. Una clase puede contener constructores sobrecargados para proporcionar los medios para inicializar los objetos de dicha clase.



Buena práctica de programación 26.6

Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse de que cada objeto se inicializa apropiadamente con valores significativos.

Cuando se crea un objeto de una clase, los *inicializadores* pueden proporcionarse entre paréntesis a la derecha del nombre de la clase. Estos inicializadores se pasan como argumentos al constructor de la clase. En el siguiente ejemplo demostraremos esta técnica. También hemos visto esta técnica varias veces antes, cuando creamos nuevos objetos de clases como **DecimalFormat**, **JLabel**, **JTextField**, **JTextArea** y **JButton**. Para cada una de estas clases vimos instrucciones de la forma

```
ref = new NombreClase( argumentos );
```

en donde **ref** es una referencia al tipo de dato apropiado; **new** indica la creación del nuevo objeto; **NombreClase** indica el tipo del nuevo objeto, y **argumentos** especifica los valores utilizados por el constructor de la clase para inicializar al objeto.

Si no se definen constructores para la clase, el compilador crea un *constructor predeterminado* que no toma argumentos (también llamado *constructor sin argumentos*). El *constructor predeterminado* de una clase

llama al constructor predeterminado de la clase a la cual extiende, luego procede a inicializar las variables de instancia de la manera en que explicamos anteriormente (es decir, las variables numéricas primitivas en 0, las booleanas en `false` y las referencias en `null`). Si la clase que extiende a esta clase no contiene un constructor predeterminado, el compilador emite un mensaje de error. También es posible que los programadores proporcionen un constructor sin argumentos como lo mostramos con la clase `Hora1` y que veremos en el siguiente ejemplo. Si el programador define un constructor, Java no creará el constructor predeterminado para la clase.



Error común de programación 26.4

Si se proporcionan los constructores para la clase, pero ninguno de los constructores públicos es un constructor sin argumentos, y se intenta hacer una llamada al constructor sin argumentos para inicializar un objeto de la clase, ocurre un error de sintaxis. Es posible llamar a un constructor sin argumentos solamente si no existen constructores para esa clase (se llama al constructor predeterminado), o si no existe un constructor sin argumentos.

26.6 Uso de los métodos *obtener* y *establecer*

Las variables de instancia privadas pueden manipularse únicamente a través de los métodos de la clase. Una manipulación común podría ser el ajuste del saldo de un cliente en el banco (por ejemplo, una variable de instancia de la clase `CuentaBanco`) por medio de un método `calculaInteres`.

Con frecuencia, las clases proporcionan métodos públicos para permitir a los clientes de la clase *establecer* u *obtener* variables de instancia privadas. Estos métodos no necesitan llamarse *establecer* u *obtener*, pero con frecuencia se llaman así. Si usted realiza un estudio más profundo de Java verá que la convención de nombres es importante para crear componentes de software reutilizable en llamados *JavaBeans*.

Como un ejemplo de nomenclatura, un método que establece la variable de instancia `tasaInteres` por lo general se escribiría como `estableceTasaInteres` y el método que obtiene `tasaInteres` por lo general se llamaría `obtieneTasaInteres`. Por lo general, a los métodos *obtener* también se les conoce como *métodos de acceso* o *métodos de consulta*. Por lo general, a los métodos *establecer* también se les conoce como *métodos de mutación* (debido a que por lo general modifican un valor).

Podría parecer que proporcionar las capacidades de las funciones *obtener* y *establecer* es, en esencia, lo mismo que hacer públicas las variables de instancia. Ésta es otra sutileza de Java que hace al lenguaje tan apropiado para la ingeniería de software. Si una variable de instancia es pública, puede leerse o escribirse en dicha variable de instancia por medio de cualquier método del programa. Si una variable de instancia es privada, ciertamente parecería que un método *obtener* permitiría a otros métodos leer sus datos, pero el método *obtener* controla el formato y el despliegado de los datos. Un método *establecer* público puede, y muy probablemente lo hará, intentar hacer un cuidadoso escrutinio para modificar el valor de la variable de instancia. Esto garantiza que el nuevo valor es apropiado para dicho elemento de dato. Por ejemplo, intentar *establecer* un día del mes para una fecha con día 37 será rechazado, intentar *establecer* el peso de una persona en un valor negativo será rechazado, y así sucesivamente. Por lo tanto, aunque los métodos *establecer* y *obtener* pueden proporcionar acceso a datos privados, el programador restringe el acceso por medio de la implementación de los métodos.

Los beneficios de la integridad de datos no son automáticos sencillamente porque las variables de instancia se hagan privadas; el programador debe proporcionar las validaciones necesarias. Java proporciona el marco de trabajo en el que los programadores pueden diseñar mejores programas de manera más conveniente.



Observación de ingeniería de software 26.11

Los métodos que establecen los valores de datos privados deben verificar que los nuevos valores que se pretenden sean apropiados; si no lo son, los métodos establecer deben colocar las variables de instancia privadas en un estado consistente apropiado.

Los métodos *establecer* de una clase pueden devolver valores que indiquen que se hicieron intentos para asignar datos no válidos a los objetos de la clase. Esto permite a los clientes de la clase verificar los valores de retorno de los métodos *establecer* para determinar si los objetos que manipulan son válidos, y tomar la decisión adecuada si no lo son.



Buena práctica de programación 26.7

Todo método que modifica las variables de instancia privadas de un objeto deben asegurarse de que los datos permanecen en un estado consistente.

El applet de la figura 26.3 mejora nuestra clase **Hora** (ahora llamada **Hora2**) para que incluya los métodos *obtener* y *establecer* para las variables de instancia privadas **hora**, **minuto** y **segundo**. Los métodos *establecer* controlan de manera estricta el establecimiento de las variables de instancia en valores válidos. Intentar establecer una variable de instancia en un valor incorrecto provoca que la variable de instancia se establezca en cero (y la deja en un estado consistente). Cada método *obtener* simplemente devuelve el valor apropiado de las variables de instancia. Este applet además introduce las técnicas avanzadas de manipulación de eventos GUI al comenzar con la definición de nuestra primera aplicación completa con ventanas.

```
1 // Figura 26.3: Hora2.java
2 // Definición de la clase Hora2
3 package com.deitel.chtp4.Cap26;      // coloca a Hora2 en un paquete
4 import java.text.DecimalFormat;        // utilizado para dar formato al número
5
6 // Esta clase mantiene la hora en formato de 24 horas
7 public class Hora2 extends Object {
8     private int hora;          // 0 - 23
9     private int minuto;        // 0 - 59
10    private int segundo;       // 0 - 59
11
12    // El constructor Hora2 inicializa en cero a cada
13    // variable de instancia. Garantiza que el objeto Hora inicia en un
14    // estado consistente.
15    public Hora2() { estableceHora( 0, 0, 0 ); }
16
17    // Métodos establecer
18    // Establece un nuevo valor de hora por medio del horario universal.
19    // Realiza validaciones de datos. Establece en cero a los valores
20    // inválidos.
21    public void estableceHora( int h, int m, int s )
22    {
23        estableceHora( h );           // establece la hora
24        estableceMinuto( m );        // establece el minuto
25        estableceSegundo( s );       // establece el segundo
26    } // fin del método estableceHora
27
28    // establece la hora
29    public void estableceHora( int h )
30    {
31        hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
32
33    // establece el minuto
34    public void estableceMinuto( int m )
35    {
36        minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
37
38    // establece el segundo
39    public void estableceSegundo( int s )
40    {
41        segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
42
43    // Métodos obtener
44    // obtiene la hora
45    public int obtieneHora() { return hora; }
46
47    // obtiene el minuto
48    public int obtieneMinuto() { return minuto; }
49
50    // obtiene el segundo
```

Figura 26.3 Uso de los métodos *establecer* y *obtener*: **Horas2.java**. (Parte 1 de 2).

```

47     public int obtieneSegundo() { return segundo; }
48
49     // Convierte a String en hora en formato universal
50     public String aCadenaUniversal()
51     {
52         DecimalFormat dosDigitos = new DecimalFormat( "00" );
53
54         return dosDigitos.format( obtieneHora() ) + ":" +
55             dosDigitos.format( obtieneMinuto() ) + ":" +
56             dosDigitos.format( obtieneSegundo() );
57     } // fin del método aCadenaUniversal
58
59     // Convierte a String en hora en formato estándar
60     public String toString()
61     {
62         DecimalFormat dosDigitos = new DecimalFormat( "00" );
63
64         return ( ( obtieneHora() == 12 || obtieneHora() == 0 ) ?
65                 12 : obtieneHora() % 12 ) + ":" +
66             dosDigitos.format( obtieneMinuto() ) + ":" +
67             dosDigitos.format( obtieneSegundo() ) +
68             ( obtieneHora() < 12 ? " AM" : " PM" );
69     } // fin del método toString
70 } // fin de la clase Hora2

```

Figura 26.3 Uso de los métodos establecer y obtener: **Hora2.java**. (Parte 2 de 2.)

```

71 // Figura 26.3: PruebaHora.java
72 // Demostración de los métodos establecer y obtener de la clase Hora2
73 import java.awt.*;
74 import java.awt.event.*;
75 import javax.swing.*;
76 import com.deitel.chtp4.Cap26.Hora2;
77
78 public class PruebaHora extends JApplet
79             implements ActionListener {
80     private Hora2 h;
81     private JLabel etiquetaHora, etiquetaMinuto, etiquetaSegundo;
82     private JTextField campoHora, campoMinuto,
83                     campoSegundo, despliega;
84     private JButton botonMarcar;
85
86     public void init()
87     {
88         h = new Hora2();
89
90         Container c = getContentPane();
91
92         c.setLayout( new FlowLayout() );
93         etiquetaHora = new JLabel( "Establece la hora" );
94         campoHora = new JTextField( 10 );
95         campoHora.addActionListener( this );
96         c.add( etiquetaHora );
97         c.add( campoHora );
98

```

Figura 26.3 Uso de los métodos establecer y obtener: **PruebaHora.java**. (Parte 1 de 4.)

```
99         etiquetaMinuto = new JLabel( "Establece los minutos" );
100        campoMinuto = new JTextField( 10 );
101        campoMinuto.addActionListener( this );
102        c.add( etiquetaMinuto );
103        c.add( campoMinuto );
104
105        etiquetaSegundo = new JLabel( "Establece los segundos" );
106        campoSegundo = new JTextField( 10 );
107        campoSegundo.addActionListener( this );
108        c.add( etiquetaSegundo );
109        c.add( campoSegundo );
110
111        despliega = new JTextField( 30 );
112        despliega.setEditable( false );
113        c.add( despliega );
114
115        botonMarcar = new JButton( "Agrega 1 a segundo" );
116        botonMarcar.addActionListener( this );
117        c.add( botonMarcar );
118
119        actualizadespliega();
120    } // fin del método init
121
122    public void actionPerformed( ActionEvent e )
123    {
124        if ( e.getSource() == botonMarcar )
125            marca();
126        else if ( e.getSource() == campoHora ) {
127            h.estableceHora(
128                Integer.parseInt( e.getActionCommand() ) );
129            campoHora.setText( "" );
130        }
131        else if ( e.getSource() == campoMinuto ) {
132            h.estableceMinuto(
133                Integer.parseInt( e.getActionCommand() ) );
134            campoMinuto.setText( "" );
135        }
136        else if ( e.getSource() == campoSegundo ) {
137            h.estableceSegundo(
138                Integer.parseInt( e.getActionCommand() ) );
139            campoSegundo.setText( "" );
140        }
141
142        actualizadespliega();
143    } // fin del método actionPerformed
144
145    public void actualizadespliega()
146    {
147        despliega.setText( "Hora: " + h.obtieneHora() +
148            "; Minuto: " + h.obtieneMinuto() +
149            "; Segundo: " + h.obtieneSegundo() );
150        showStatus( "La hora estandar es: " + h.toString() +
151            "; La hora universal es: " + h.aCadenaUniversal() );
152    } // fin del método actualizadespliega
153
154    public void marca()
```

Figura 26.3 Uso de los métodos establecer y obtener: **PruebaHora.java**. (Parte 2 de 4.)

```

155    {
156        h.estableceSegundo( ( h.obtieneSegundo() + 1 ) % 60 );
157
158        if ( h.obtieneSegundo() == 0 ) {
159            h.estableceMinuto( ( h.obtieneMinuto() + 1 ) % 60 );
160
161            if ( h.obtieneMinuto() == 0 )
162                h.estableceHora( ( h.obtieneHora() + 1 ) % 24 );
163        } // fin de if
164    } // fin del método marca
165 } // fin de la clase PruebaHora

```

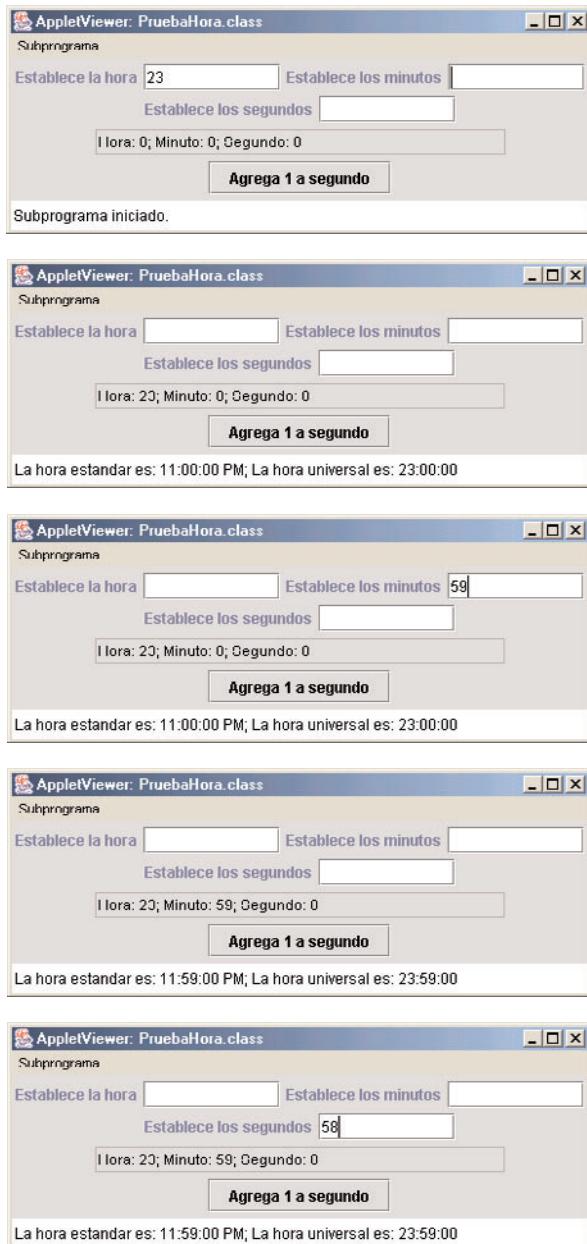


Figura 26.3 Uso de los métodos *establecer* y *obtener*; **PruebaHora.java**. (Parte 3 de 4.)

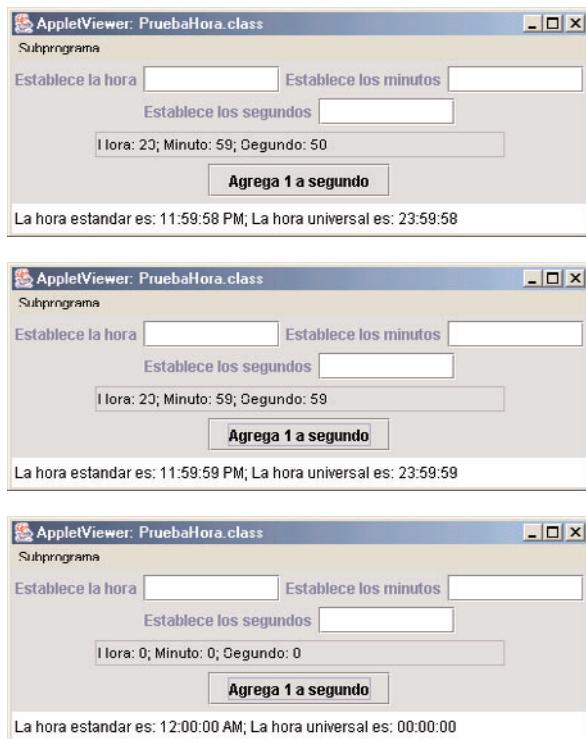


Figura 26.3 Uso de los métodos *establecer* y *obtener*; **PruebaHora.java**. (Parte 4 de 4.)

Los nuevos métodos *establecer* de la clase se definen en las líneas 28, 32 y 36 respectivamente. Observe que cada método realiza la misma instrucción condicional que estaba previamente en el método **estableceHora** para establecer la **hora**, el **minuto** y el **segundo**. Con la adición de estos métodos fuimos capaces de redefinir el cuerpo del método **estableceHora** (línea 20) para utilizar estos tres métodos y establecer la hora.

Observación de ingeniería de software 26.12



Si un método de la clase proporciona toda o parte de la funcionalidad requerida por otro método de la clase, lláme a dicho método desde otro método. Esto simplifica el mantenimiento del código y reduce la probabilidad de error si la implementación del código se modifica. También es un ejemplo claro de la reutilización.

Debido a las modificaciones en la clase **Hora2** que describimos antes, minimizamos las modificaciones que tienen que llevarse a cabo en la definición de la clase si la representación de los datos se modifica de **hora**, **minuto**, **segundo** a otra representación (tal como los segundos transcurridos durante el día). Solamente será necesario modificar los cuerpos de los métodos *establecer* y *obtener*. Esto permite al programador modificar la implementación de la clase sin afectar a los clientes de la misma clase (mientras los métodos públicos de la clase se llamen de la misma manera).

El applet **PruebaHora** proporciona una interfaz gráfica de usuario que permite al usuario ejecutar los métodos de la clase **Hora2**. El usuario puede establecer el valor de la hora, el minuto o el segundo al escribir un valor en el **JTextField** y oprimir la tecla *Entrar*. El usuario también puede hacer clic en el botón **Agrega 1 a segundo** para incrementar el tiempo en un segundo. En este applet, todos los eventos **JTextField** y **JButton** se procesan en el método **actionPerformed** (línea 122). Observe que las líneas 95, 101, 107 y 116 llaman a **addActionListener** para indicar que el applet debe comenzar a poner atención a **campoHora**, **campoMinuto**, **campoSegundo** de tipo **JTextField**, y a **botonMarcar** de tipo **JButton**, respectivamente. Además, observe que las cuatro llamadas utilizan **this** como argumento, lo que indica que el objeto de nuestra clase applet **PruebaHora** invoca a su **actionPerformed** para cada interacción con el usuario con estos compo-

nentes GUI. Esto provoca la siguiente interesante pregunta, ¿cómo determinamos el componente GUI con el que interactuó el usuario?

En **actionPerformed**, observe el uso de **e.getSource()** para determinar cuál componente GUI generó el evento. Por ejemplo, en la línea 124

```
if ( e.getSource() == botonMarcar )
```

determina si el usuario hizo clic en **botonMarcar**. Si es así, se ejecuta el cuerpo de la estructura **if**. De lo contrario, se evalúa la condición de la estructura **if** correspondiente a la línea 126, etcétera. Todo evento tiene una *fuente*, el componente GUI con el que el usuario interactuó para señalar al programa que realice una tarea. El parámetro **ActionEvent** que se le proporcionó a **actionPerformed** cada vez que ocurre el evento contiene una referencia hacia la fuente. La condición anterior simplemente pregunta, “¿la fuente del evento es **botonMarcar**?“

Después de cada operación, se despliega la hora resultante como una cadena, en la barra de estado del applet. Las ventanas de salida muestran al applet antes y después de las siguientes operaciones: establecer la hora en 23, establecer el minuto en 59, establecer el segundo en 58, e incrementar el segundo al doble con el botón **Agrega 1 a segundo**.

Observe que cuando se hace clic en el botón **Agrega 1 a segundo**, el método **actionPerformed** llama al método **marcar** (línea 154) del applet. El método **marcar** utiliza todos los nuevos métodos **obtener** y **establecer** para incrementar de manera apropiada los segundos. Aunque esto funciona, incurre en la sobrecarga de llamadas a múltiples métodos.

Error común de programación 26.5



Un constructor puede llamar a otros métodos de la clase, tal como los métodos establecer y obtener, pero debido a que el constructor inicializa el objeto, las variables de instancia no pueden aún estar en un estado consistente. El uso de las variables de instancia antes de inicializarse de manera apropiada, es un error.

Es verdad que los métodos *establecer* son importantes desde el punto de vista de la ingeniería de software, ya que pueden realizar validaciones. Los métodos *establecer* y *obtener* tienen otra ventaja en la ingeniería de software, como lo explicamos en la siguiente *Observación de ingeniería de software*.

Observación de ingeniería de software 26.13



*Acceder a los datos **private** a través de los métodos establecer y obtener no solamente protege a las variables de instancia de recibir valores no válidos, sino que además aísla a los clientes de la clase de la representación de las variables de instancia. Por lo tanto, si la representación de los datos cambia (por lo general, para reducir el almacenamiento requerido, o para mejorar el rendimiento), solamente necesita modificar las implementaciones del método; los clientes no necesitan modificación alguna mientras la interfaz proporcionada por los métodos permanezca igual.*

26.7 Uso de la referencia this

Cuando el método de una clase hace referencia a otro miembro de dicha clase para un objeto específico de la misma clase, ¿cómo asegura Java que se hace referencia al objeto apropiado? La respuesta es que cada objeto tiene acceso a una referencia a sí mismo, llamada *referencia this*.

La referencia **this** se utiliza implícitamente para hacer referencia tanto a las variables de instancia como a los métodos de un objeto. Por ahora, mostramos un ejemplo sencillo del uso explícito de la referencia **this**; más adelante, mostraremos algunos ejemplos sustanciales y sutiles del uso de **this**.

Tip de rendimiento 26.2



Java conserva el almacenamiento, manteniendo sólo una copia de cada método por clase; este método es invocado por cada objeto de dicha clase. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase.

La aplicación de la figura 26.4 muestra el uso implícito y explícito de la referencia **this** para permitir al método **main** de la clase **PruebaThis** desplegar los datos **private** del objeto **HorasSimple**.

La clase **HorasSimple** (líneas 20 a 46) define tres variables de instancia privadas, **hora**, **minuto** y **segundo**. El constructor (línea 23) recibe tres argumentos **int** para inicializar un objeto **HorasSimple**. Ob-

```
1 // Figura 26.4: PruebaThis.java
2 // Uso de la referencia this para hacer referencia a
3 // las variables de instancia y a los métodos.
4 import javax.swing.*;
5 import java.text.DecimalFormat;
6
7 public class PruebaThis {
8     public static void main( String args[] )
9     {
10         HoraSimple h = new HoraSimple( 12, 30, 19 );
11
12         JOptionPane.showMessageDialog( null, h.construyeCadena(),
13             "Demostracion de la referencia \"this\" ",
14             JOptionPane.INFORMATION_MESSAGE );
15
16         System.exit( 0 );
17     } // fin del método main
18 } // fin de la clase PruebaThis
19
20 class HoraSimple {
21     private int hora, minuto, segundo;
22
23     public HoraSimple( int hora, int minuto, int segundo )
24     {
25         this.hora = hora;
26         this.minuto = minuto;
27         this.segundo = segundo;
28     } // fin del constructor HoraSimple
29
30     public String construyeCadena()
31     {
32         return "this.toString(): " + this.toString() +
33             "\ntoString(): " + toString() +
34             "\nthis (con una llamada implícita a toString()): " +
35             this;
36     } // fin del método construyeCadena
37
38     public String toString()
39     {
40         DecimalFormat dosDigitos = new DecimalFormat( "00" );
41
42         return dosDigitos.format( this.hora ) + ":" +
43             dosDigitos.format( this.minuto ) + ":" +
44             dosDigitos.format( this.segundo );
45     } // fin del método toString
46 } // fin de la clase HoraSimple
```

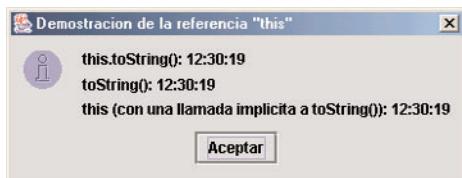


Figura 26.4 Uso de la referencia **this**.

serve que los nombres de los parámetros para el constructor son los mismos que los nombres de las variables de instancia. Recuerde que una variable local de un método con el mismo nombre que una variable de instancia de la clase, oculta la variable de instancia en el alcance del método. Por esta razón, utilizamos la referencia **this** para hacer referencia explícita a la variable de instancia de las líneas 25 a 27.



Error común de programación 26.6

*En un método en el que un parámetro del método tiene el mismo nombre que uno de los miembros de la clase, utilice explícitamente **this** si quiere tener acceso al miembro de la clase; de lo contrario, hará una referencia incorrecta al parámetro del método.*



Buena práctica de programación 26.8

Evite utilizar nombres de parámetros que tengan conflictos con los nombres de los métodos de las clases.

El método **construyeCadena** (líneas 30 a 36) devuelve una **String** creada mediante la instrucción

```
return "this.toString(): " + this.toString() +
       "\ntoString(): " + toString() +
       "\nthis (con una llamada implícita a toString()): " +
       this;
```

la cual utiliza la referencia **this** de tres maneras. La primera línea invoca explícitamente el método **toString** de la clase, por medio de **this.toString()**. La segunda línea utiliza de manera implícita la referencia **this** para realizar la misma tarea. La tercera línea agrega **this** a la cadena que será devuelta. Recuerde que la referencia **this** es una referencia a un objeto; el objeto **HoraSimple** actual que se manipula. Como antes, cualquier referencia que se agrega a **String** da como resultado una llamada al método **toString** para el objeto referenciado. En la línea 12 se invoca el método **construyeCadena** para desplegar el resultado de las tres llamadas a **toString**. Observe que se despliega la misma hora en las tres líneas de salida, ya que las tres llamadas a **toString** son para el mismo objeto.

26.8 Finalizadores

Ya vimos que los métodos constructores son capaces de inicializar los datos de un objeto de la clase, cuando ésta se crea. Por lo general, los constructores adquieren recursos de sistema tales como memoria (cuando utiliza el comando **new**). Necesitamos una manera disciplinada de devolver los recursos al sistema cuando ya no son necesarios, para evitar el agotamiento de recursos. El recurso que más solicitan los constructores es la memoria. Java realiza la *recolección automática de basura* en la memoria para ayudar a devolver la memoria al sistema. Cuando un objeto ya no se utiliza en el programa (es decir, no existen referencias hacia el objeto), el objeto se *marca para el recolector de basura*. La memoria para dicho objeto puede reclamarse cuando se ejecuta el *recolector de basura*. Por tal motivo, las fugas de memoria que son comunes en otros lenguajes como C y C++ (debido a que la memoria no se reclama de manera automática en dichos lenguajes) son menos comunes en Java. Sin embargo, pueden ocurrir otras fugas de recursos.

Todas las clases en Java pueden tener un *método finalizador* que devuelva los recursos al sistema. Con seguridad, el método finalizador de un objeto será llamado para realizar la *limpieza final* en el objeto, justo antes de que el recolector de basura reclame la memoria del objeto. Un método finalizador de la clase siempre tiene el nombre **finalize**, no recibe parámetros y no devuelve valor alguno (es decir, su tipo de retorno es **void**). Una clase sólo puede tener un método **finalize** que no toma argumentos. El método **finalize** está definido originalmente en la clase **Object**, como un contenedor que no realiza acción alguna. Esto garantiza que cada clase tiene un método **finalize** para llamar al recolector de basura.

Hasta aquí, no hemos proporcionado finalizadores para las clases que hemos explicado. En realidad, los finalizadores rara vez se utilizan con clases sencillas. Veremos un ejemplo del método **finalize**, y explicaremos el recolector de basura más adelante en la figura 26.5.

26.9 Miembros estáticos de una clase

Cada objeto de una clase tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe compartirse una copia de una variable en particular entre todos los objetos de la clase. Una *variable*

de clase **static** se utiliza por ésta y por otras razones. Una variable de clase **static** representa información para toda la clase; todos los objetos de la clase comparten las mismas piezas de datos. La declaración de un miembro **static** comienza con la palabra reservada **static**.

Motivemos la necesidad de datos **static** para toda una clase con un ejemplo de juego de video. Suponga que tenemos un juego de video con **Marcianos** y otras criaturas del espacio. Cada **Marciano** tiende a ser más valiente y está más dispuesto a atacar a otras criaturas del espacio cuando el **Marciano** se da cuenta de que existen al menos cinco **Marcianos** presentes. Si existen menos de cinco **Marcianos** presentes, cada **Marciano** se acobarda. De modo que cada **Marciano** necesita conocer la **cuentaMarcianos**. Incluiremos a la clase **Marciano** el dato **cuentaMarcianos** como un dato de instancia. Si hacemos esto, entonces cada **Marciano** tendrá una copia separada del dato de instancia cada vez que creemos un nuevo **Marciano**, y no tendremos que actualizar la variable de instancia **cuentaMarcianos** en cada **Marciano**. Esto desperdicia espacio con copias redundantes y desperdicia tiempo en actualizar las copias separadas. En vez de lo anterior declaramos **cuentaMarcianos** como **static**. Esto hace a **cuentaMarcianos** un dato para toda la clase. Cada **Marciano** puede ver a **cuentaMarciano** como si fuera un dato de instancia del **Marciano**, pero solamente se mantiene una copia del **cuentaMarcianos** de tipo **static** en Java. Esto ahorra espacio. Ahorramos tiempo al incrementar **cuentaMarcianos static** del constructor de **Marciano**. Sólo existe una copia, de modo que no tenemos que incrementar copias separadas de **cuentaMarcianos** para cada objeto **Marciano**.

Tip de rendimiento 26.3



Utilice las variables de clase **static** para ahorrar espacio, cuando sea suficiente una sola copia de los datos.

Aunque las variables de clase **static** pueden parecer como variables globales, las variables de clase **static** tienen alcance de clase. Se puede acceder a los miembros de clase **public static** de la clase a través de una referencia a cualquier copia de la clase, o se puede acceder a ellas a través del nombre de la clase por medio del operador punto (por ejemplo, **Math.random()**). Se puede acceder a los miembros de clase **private static** de la clase a través de los métodos de la misma clase. En realidad, los miembros de clase **static** existen incluso cuando no existen objetos de dicha clase; están disponibles tan pronto como la clase se carga dentro de la memoria en tiempo de ejecución. Para acceder a un miembro de clase **private static** cuando no existen objetos de la clase, debe proporcionarse un método **public static** y el método debe invocarse colocando como prefijo el nombre de la clase y el operador punto.

El programa de la figura 26.5 muestra el uso de las variables de clase **private static** y de un método **public static**. La variable de clase **cuenta** se inicializa en cero de manera predeterminada. La variable de clase **cuenta** mantiene una cuenta del número de objetos de la clase **Empleado** que se instancia, y que actualmente reside en memoria. Esto incluye objetos que ya están señalados para el recolector de basura pero que aún no son reclamados.

```

1 // Figura 26.5: Empleado.java
2 // Declaración de la clase Empleado.
3 public class Empleado extends Object {
4     private String nombre;
5     private String apellido;
6     private static int cuenta; // # de objetos en memoria
7
8     public Empleado( String nomb, String apell )
9     {
10         nombre = nomb;
11         apellido = apell;
12
13         ++cuenta; // incrementa la cuenta estática de empleados
14         System.out.println( "Constructor del objeto Empleado: " +

```

Figura 26.5 Uso de una variable de clase **static** para mantener la cuenta del número de objetos de una clase; **Empleado.java**. (Parte 1 de 2.)

```

15                     nombre + " " + apellido );
16     } // fin del constructor Empleado
17
18     protected void finalize()
19     {
20         --cuenta; // disminuye la cuenta estática de empleados
21         System.out.println( "Finalizador del objeto Empleado: " +
22                             nombre + " " + apellido +
23                             "; cuenta = " + cuenta );
24     } // fin del método finalize
25
26     public String obtieneNombre() { return nombre; }
27
28     public String obtieneApellido() { return apellido; }
29
30     public static int obtieneCuenta() { return cuenta; }
31 } // fin de la clase Empleado

```

Figura 26.5 Uso de una variable de clase **static** para mantener la cuenta del número de objetos de una clase; **Empleado.java**. (Parte 2 de 2.)

```

32 // Figura 26.5: PruebaEmpleado.java
33 // Prueba la clase empleado con una variable de clase estática,
34 // con un método de clase estática, y con memoria dinámica.
35 import javax.swing.*;
36
37 public class PruebaEmpleado {
38     public static void main( String args[] )
39     {
40         String salida;
41
42         salida = "Empleados antes de crear la instancia: " +
43                 Empleado.obtieneCuenta();
44
45         Empleado e1 = new Empleado( "Susana", "Baez" );
46         Empleado e2 = new Empleado( "Roberto", "Jimenez" );
47
48         salida += "\n\nEmpleados despues de crear la instancia: " +
49                 "\nvia e1.obtieneCuenta(): " + e1.obtieneCuenta() +
50                 "\nvia e2.obtieneCuenta(): " + e2.obtieneCuenta() +
51                 "\nvia Empleado.obtieneCuenta(): " +
52                 Empleado.obtieneCuenta();
53
54         salida += "\n\nEmpleado 1: " + e1.obtieneNombre() +
55                 " " + e1.obtieneApellido() +
56                 "\nEmpleado 2: " + e2.obtieneNombre() +
57                 " " + e2.obtieneApellido();
58
59         // marca los objetos a los que hace referencia e1 y e2
60         // para recolección de basura
61         e1 = null;
62         e2 = null;
63
64         System.gc(); // sugiere llamar al recolector de basura

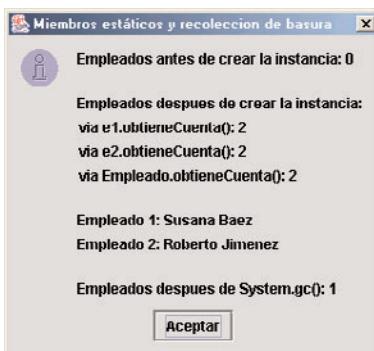
```

Figura 26.5 Uso de una variable de clase **static** para mantener la cuenta del número de objetos de una clase; **PruebaEmpleado.java**. (Parte 1 de 2.)

```

65         salida += "\n\nEmpleados despues de System.gc(): " +
66             Empleado.obtieneCuenta();
67
68         JOptionPane.showMessageDialog( null, salida,
69             "Miembros estáticos y recolección de basura",
70             JOptionPane.INFORMATION_MESSAGE );
71         System.exit( 0 );
72     } // fin de main
73 } // fin de la clase PruebaEmpleado

```



```

Constructor del objeto Empleado: Susana Baez
Constructor del objeto Empleado: Roberto Jimenez
Finalizador del objeto Empleado: Susana Baez; cuenta = 1
Finalizador del objeto Empleado: Roberto Jimenez; cuenta = 0

```

Figura 26.5 Uso de una variable de clase **static** para mantener la cuenta del número de objetos de una clase; **PruebaEmpleado.java**. (Parte 2 de 2.)

Cuando existen objetos de la clase **Empleado**, el miembro **cuenta** puede utilizarse en cualquier método de un objeto **Empleado**; en este ejemplo, el constructor incrementa la **cuenta** (línea 13) y el finalizador la disminuye (línea 20). Cuando no existen objetos de la clase **Empleado**, todavía se puede hacer referencia al miembro **cuenta**, pero solamente a través de una llamada al método **public static obtieneCuenta** de la siguiente manera:

```
Empleado.obtieneCuenta()
```

En este ejemplo, el método **obtieneCuenta** determina el número de objetos **Empleado** actualmente en memoria. Observe que cuando no existen objetos instanciados en el programa, se emite la llamada al método **Empleado.obtieneCuenta()**. Sin embargo, cuando existen instancias de objetos, el método **obtieneCuenta** también puede invocarse a través de una referencia a uno de los objetos, como en

```
e1.obtieneCuenta()
```



Buena práctica de programación 26.9

*Siempre invoque métodos **static** por medio del nombre de la clase y del operador punto (.). Esto enfatiza a otros programadores que leen su código que el método que llaman es un método **static**.*

Observe que la clase **Empleado** tiene un método **finalize** (línea 18). Este método se incluye para mostrar cuándo se llama al recolector de basura en un programa. Por lo general, el método **finalize** se declara como **protected**, de modo que no es parte de los servicios **public** de la clase. Explicaremos el modificador de acceso **protected** con detalle en el capítulo 27.

El método **main** de la aplicación **PruebaEmpleado** crea dos instancias del objeto **Empleado** (líneas 45 y 46). Cuando se invocan cada uno de los constructores del objeto **Empleado**, líneas 10 y 11, almacenan

referencias a los objetos **String** para el nombre y el apellido de dicho **Empleado**. Observe que estas dos instrucciones *no* hacen copias de los argumentos **String** originales. En realidad, los objetos **String** en Java son *inmutables*, éstos no pueden modificarse una vez creados (la clase **String** no proporciona método *establecer* alguno). Una referencia no puede utilizarse para modificar una **String**, de modo que es seguro hacer muchas referencias al objeto **String** en el programa en Java. Por lo general, éste no es el caso de la mayoría de las clases en Java.

Cuando **main** termina con los dos objetos **Empleado**, las referencias **e1** y **e2** se establecen en **null**, en las líneas 61 y 62. En este punto, las referencias **e1** y **e2** ya no hacen referencia a los objetos instanciados en las líneas 45 y 46. Esto marca a los objetos para el *recolector de basura*, debido a que no existen referencias a los objetos en el programa.

En algún momento, el recolector de basura reclama la memoria para estos casos (o el sistema operativo reclama la memoria cuando termina el programa). No existe certeza de cuándo actuará el recolector de basura, de modo que hacemos una llamada explícita al recolector de basura con la línea 64

```
System.gc(); // sugiere llamar al recolector de basura
```

la cual utiliza el método **public static gc** de la clase **System** (paquete **java.lang**), para sugerir la ejecución inmediata del recolector de basura. Sin embargo, ésta solamente es una sugerencia para la Java Virtual Machine (el intérprete); la sugerencia puede ignorarse. En nuestro ejemplo, el recolector de basura se ejecutó antes de que las líneas 69 a 71 desplegaran los resultados del programa. La última línea de la salida indica que el número de objetos **Empleado** en memoria es 1 después de llamar a **System.gc()**. Además, las dos últimas líneas de la salida en la ventana de comandos muestran que el objeto **Empleado** para **Susana Baez** se finalizó antes del objeto **Empleado** para **Roberto Jiménez**. El recolector de basura no garantiza la ejecución cuando se invoca a **System.gc()**, y no existe la garantía de que el recolector de basura recoja los objetos en un orden específico, de modo que es posible que la salida para este programa en su sistema puede diferir.

[Nota: Un método que se declara como **static** no puede acceder a miembros no estáticos de la clase. A diferencia de los métodos no estáticos, un método **static** no tiene referencia **this** debido a que las variables de clase estáticas y los métodos de clase están independientemente de cualquier objeto de la clase y antes de que se genere cualquier instancia de un objeto de la clase.]

Error común de programación 26.7



Hacer referencia a **this** en un método **static**, es un error de sintaxis.

Error común de programación 26.8



Es un error de sintaxis que un método **static** llame a un método de instancia o que acceda a una variable de instancia.

Observación de ingeniería de software 26.14



Cualquier variable de una clase **static** y cualquier método de una clase **static** puede utilizarse incluso si ningún objeto de esa clase se ha instanciado.

RESUMEN

- La POO encapsula los datos (atributos) y los métodos (comportamientos) dentro de objetos; los datos y los métodos de un objeto están íntimamente relacionados.
- Los objetos tienen la propiedad de ocultar la información. Los objetos pueden saber cómo comunicarse entre sí a través de interfaces bien definidas, pero por lo general no se les permite conocer la manera en que se implementan los demás objetos.
- Los programadores en Java se concentran en la creación de sus propios tipos definidos por el usuario llamados clases. A los componentes de datos de las clases se les conoce como variables de instancia.
- Java utiliza la herencia para crear nuevas clases, a partir de las definiciones de clases existentes.
- Cada clase en Java es una subclase de **Object**. Entonces, cada nueva definición de una clase tiene los atributos (datos) y comportamientos (métodos) de la clase **Object**.

- Las palabras reservadas **public** y **private** son modificadores de acceso a los datos.
- Las variables de instancia y los métodos que se declaran con el modificador de acceso a datos **public** son accesibles en donde quiera que el programa haga referencia al objeto de la clase en la que están definidos.
- Las variables de instancia y los métodos que se declaran con el modificador de acceso a datos **private** son accesibles solamente en los métodos de la clase en la que están definidos.
- Por lo general, las variables de instancia se declaran **private** y, por lo general, los métodos se declaran **public**.
- Los clientes de una clase utilizan los métodos públicos (o servicios públicos) de dicha clase para manipular los datos almacenados en los objetos de la clase.
- Un constructor es un método con el mismo nombre que el de la clase que inicializa las variables de instancia de un objeto de la clase, cuando se crea la instancia de la misma clase. Los métodos constructores pueden sobrecargarse en una clase. Los constructores pueden tomar argumentos pero no pueden devolver valor alguno.
- Los constructores y otros métodos que modifican los valores de las variables de instancia siempre deben mantener a los objetos en un estado consistente.
- El método **toString** no toma argumentos y devuelve una **String** (Cadena). El método **toString** original de la clase **Object** es un contenedor que por lo general redefine una subclase.
- Cuando se crea la instancia de un objeto, el operador **new** reserva la memoria para ese objeto, luego **new** llama al constructor de la clase para inicializar las variables de instancia del objeto.
- Si los archivos **.class** para las clases utilizadas en un programa se encuentran en el mismo directorio de la clase que las utiliza, no se requieren instrucciones **import**.
- Concatenar un **String** y cualquier objeto provoca una llamada implícita al método **toString** del objeto para convertirlo en un **String**, luego se concatenan los **Strings**.
- Dentro del alcance de una clase, los miembros de la clase de inmediato están accesibles para todos los métodos de la clase y se puede hacer referencia a ellos simplemente por su nombre. Fuera del alcance de la clase, solamente se puede acceder a los miembros de la clase a través de un “manipulador” (es decir, una referencia a un objeto de la clase).
- Si un método define una variable con el mismo nombre que una variable con alcance de clase, la variable con alcance de clase se oculta detrás de la variable con alcance de método dentro del mismo método. Se puede acceder a una variable de instancia oculta colocando antes del nombre la palabra reservada **this** y el operador punto.
- Cada clase e interfaz en el API de Java pertenece a un paquete específico que contiene un grupo de clases e interfaces relacionadas.
- En realidad, los paquetes son estructuras de directorios que se utilizan para organizar las clases y las interfaces. Los paquetes proporcionan un mecanismo para la reutilización de software y una convención para los nombres de clases únicos.
- Crear clases reutilizables requiere: definir una clase pública, agregar una instrucción **package** al archivo de definición de la clase, compilar la clase en la estructura de directorio apropiada para el paquete para tener la nueva clase disponible para el compilador y el intérprete, e importar la clase dentro de un programa.
- Java 2 tiene un directorio llamado **classes** en donde se coloca la versión compilada de algunas clases reutilizables que son bien conocidas tanto por el compilador como por el intérprete.
- Cuando compile una clase en un paquete, debe pasar la opción **-d** al compilador para especificar en dónde crear (o localizar) todos los directorios de la instrucción **package**.
- Los nombres del directorio **package** se vuelven parte del nombre de la clase cuando ésta se compila. Utilice este identificador completo en los programas, o importe las clases y utilice su nombre corto (el nombre de la clase por sí mismo) en el programa.
- Si no se definen constructores para una clase, el compilador crea un constructor predeterminado que no toma argumentos.
- Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos de la segunda clase.
- Las clases con frecuencia proporcionan métodos públicos para permitir a los clientes de la clase *establecer* (es decir, asignar valores) u *obtener* (es decir, adquirir valores) de variables de instancia privadas. Por lo general, los métodos *obtener* son conocidos como métodos de acceso o métodos de consulta. Por lo general, a los métodos *establecer* se les llama métodos mutantes (debido a que por lo general modifican un valor).
- Todo evento tiene una fuente; el componente GUI con el que el usuario interactúa para indicar al programa qué tarea realizar.
- Cuando no se proporciona ningún modificador de acceso a miembros para un método o una variable, cuando éste se define dentro de una clase, se considera que el método o la variable tienen acceso al paquete.

- Si un programa utiliza múltiples clases del mismo paquete, estas clases pueden acceder directamente a los métodos de acceso a paquetes y a los datos de los otros métodos, a través de una referencia a un objeto.
- Cada objeto tiene acceso a una referencia a sí mismo llamada referencia **this**, la cual puede utilizarse dentro de los métodos de la clase para hacer referencia explícita a los datos y objetos del objeto.
- En cualquier momento en el que usted tenga una referencia a un programa (incluso como resultado de una llamada a un método), la referencia puede ser seguida por un operador punto y una llamada a uno de los métodos del tipo de referencia.
- Todas las clases en Java pueden tener un método finalizador que devuelve los recursos al sistema. Un método finalizador de la clase siempre tiene el nombre **finalize**, no recibe parámetros y no devuelve valor alguno. El método **finalize** se define originalmente en la clase **Object** como un contenedor que no hace cosa alguna. Esto garantiza que cada clase contiene un método **finalize** para llamar al recolector de basura.
- Una variable estática de clase representa información para toda la clase; todos los objetos de la clase comparten la misma porción de información. Se puede acceder a los miembros públicos y estáticos de una clase a través de una referencia a cualquier objeto de dicha clase, o se puede acceder a ellos a través del nombre de la clase mediante el uso del operador punto.
- El método público y estático **gc** de la clase **System** sugiere que el colector de basura se ejecute inmediatamente. Esta sugerencia puede ignorarse. El recolector de basura no garantiza la recolección de todos los objetos en un orden específico.
- Un método declarado como **static** no tiene acceso a los miembros no estáticos de la clase. A diferencia de los métodos no estáticos, un método estático no contiene una referencia **this**, ya que las variables estáticas y los métodos estáticos de la clase existen independientemente de cualquier objeto de la clase.
- Los miembros estáticos de la clase existen, incluso si no existen objetos de dicha clase; éstos están disponibles tan pronto como se carga la clase en memoria en tiempo de ejecución.

TERMINOLOGÍA

acceso a paquetes	implementación de una clase	objeto
alcance de una clase	inicializar el objeto de una clase	ocultamiento de información
atributo	instancia de una clase	opción del compilador -d
biblioteca de clases	instrucción package	operador new
clase	interfaz de una clase	operador punto (.)
clase contenedora	interfaz pública de una clase	principio del menor privilegio
cliente de una clase	llamadas a métodos	private
código reutilizable	método	programación basada en objetos (PBO)
comportamiento	método de acceso	programación orientada a objetos (POO)
constructor	método de ayuda	public
constructor predeterminado	método de consulta	referencia this
constructor sin argumentos	método de instancia	reutilización de software
control de acceso a miembros	método de una clase	servicios de una clase
crear la instancia (instanciar) un objeto de una clase	(static)	tipo de dato
definición de clase	método de utilidad	tipo de dato abstracto (ADT)
encapsulamiento	método <i>establecer</i>	tipo definido por el programador
estado consistente de una variable de instancia	método mutante	tipo definido por el usuario
extender	método <i>obtener</i>	variable de clase
extensibilidad	método predicado	variable de clase static
finalizador	método static	variable de instancia
	modificadores de acceso a miembros	

ERRORES COMUNES DE PROGRAMACIÓN

- 26.1 Definir más de una clase pública en el mismo archivo, es un error de sintaxis.
- 26.2 El hecho de que un método que no es un miembro de una clase en particular intente acceder a un miembro privado de dicha clase, es un error de sintaxis.
- 26.3 Intentar declarar un tipo de retorno para un constructor y/o intentar devolver un valor desde un constructor, es un error lógico. Java permite a otros métodos de la clase tener el mismo nombre de la clase y especificar los tipos de

retorno. Dichos métodos no son constructores y no se les llamará cuando se genere la instancia de un objeto de la clase.

- 26.4 Si se proporcionan los constructores para la clase, pero ninguno de los constructores públicos es un constructor sin argumentos, y se intenta hacer una llamada al constructor sin argumentos para inicializar un objeto de la clase, ocurre un error de sintaxis. Es posible llamar a un constructor sin argumentos solamente si no existen constructores para esa clase (se llama al constructor predeterminado), o si no existe un constructor sin argumentos.
- 26.5 Un constructor puede llamar a otros métodos de la clase, tal como los métodos *establecer* y *obtener*, pero debido a que el constructor inicializa el objeto, las variables de instancia no pueden aún estar en un estado consistente. El uso de las variables de instancia antes de inicializarse de manera apropiada, es un error.
- 26.6 En un método en el que un parámetro del método tiene el mismo nombre que uno de los miembros de la clase, utilice explícitamente **this** si quiere tener acceso al miembro de la clase; de lo contrario, hará una referencia incorrecta al parámetro del método.
- 26.7 Hacer referencia a **this** en un método **static**, es un error de sintaxis.
- 26.8 Es un error de sintaxis que un método **static** llame a un método de instancia o que acceda a una variable de instancia.

BUENAS PRÁCTICAS DE PROGRAMACIÓN

- 26.1 Agrupe los miembros de acuerdo con los modificadores de acceso a miembros dentro de la definición de una clase, para mayor claridad y legibilidad.
- 26.2 Nosotros preferimos listar primero a las variables de instancia **private** de una clase, para que conforme lea el código, vea los nombres y los tipos de dichas variables, antes de utilizarlas en los métodos de la clase.
- 26.3 A pesar del hecho de que los miembros públicos y privados pueden repetirse y mezclarse, primero liste en un grupo a todos los miembros privados de la clase, y después liste en otro grupo a todos los miembros públicos.
- 26.4 Siempre defina una clase de manera que sus variables de instancia se mantengan en un estado consistente.
- 26.5 Inicialice las variables de instancia de una clase en el constructor de esa clase.
- 26.6 Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse de que cada objeto se inicializa apropiadamente con valores significativos.
- 26.7 Todo método que modifica las variables de instancia privadas de un objeto deben asegurarse de que los datos permanecen en un estado consistente.
- 26.8 Evite utilizar nombres de parámetros que tengan conflictos con los nombres de los métodos de las clases.
- 26.9 Siempre invoque métodos **static** por medio del nombre de la clase y del operador punto (.). Esto enfatiza a otros programadores que leen su código que el método que llaman es un método **static**.

TIPS DE RENDIMIENTO

- 26.1 Todos los objetos en Java se pasan por referencia. Sólo se pasa la dirección de memoria, no una copia de todo el objeto (como se haría en un paso por valor).
- 26.2 Java conserva el almacenamiento, manteniendo sólo una copia de cada método por clase; este método es invocado por cada objeto de dicha clase. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase.
- 26.3 Utilice las variables de clase **static** para ahorrar espacio, cuando sea suficiente una sola copia de los datos.

OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 26.1 Es importante escribir programas que sean claros y fáciles de mantener. La regla es el cambio, en lugar de la excepción. Los programadores deben prever que su código será modificado. Como veremos pronto, las clases facilitan la modificación de un programa.
- 26.2 Las definiciones de las clases que comienzan con la palabra reservada **public** deben almacenarse en un archivo que tiene el mismo nombre que la clase, y terminar con la extensión de archivo **.java**.
- 26.3 Toda clase definida en Java debe ser una extensión de otra clase. Si la clase no utiliza explícitamente la palabra reservada **extends** en su definición, esta clase implícitamente se extiende de **Object**.

- 26.4** Mantenga privadas todas las variables de instancia. Cuando sea necesario, proporcione métodos públicos para establecer los valores de variables de instancia privadas y para obtener los valores de variables de instancia privadas. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual reduce los errores y mejora la posibilidad de modificación del programa.
- 26.5** Los métodos tienden a caer en diversas categorías: métodos que obtienen los valores a partir de variables de instancia privadas; métodos que establecen los valores de las variables de instancia privadas; métodos que implementan los servicios de la clase; y métodos que realizan distintos mecanismos para la clase, tales como la inicialización de los objetos de las clases, la asignación de los objetos de las clases, y la conversión entre clases y los tipos predefinidos, o entre clases y otras clases.
- 26.6** Cada vez que `new` crea un objeto de la clase, se llama al constructor de dicha clase para inicializar las variables de instancia del nuevo objeto.
- 26.7** El ocultamiento de información promueve la capacidad de modificación del programa y simplifica la percepción de los clientes respecto a la clase.
- 26.8** Los clientes de una clase pueden (y deben) utilizar la clase sin conocer los detalles de implementación de la clase. Si cambia la implementación de la clase (por ejemplo, para mejorar el rendimiento), la interfaz proporcionada permanece constante, el código fuente de los clientes de la clase no necesitan modificación. Esto hace mucho más fácil la modificación de los sistemas.
- 26.9** Con frecuencia, utilizar un método de programación orientada a objetos simplifica las llamadas a los métodos, al reducir el número de parámetros a pasar. Este beneficio de la programación orientada a objetos se deriva del hecho de que el encapsulamiento de las variables de instancia y de los métodos dentro de un objeto le da a los métodos el derecho de acceso a las variables de instancia.
- 26.10** Un archivo de código fuente en Java tiene el siguiente orden: una instrucción `package` (si existe alguna), instrucción `import` (si existen), y las definiciones de las clases. Solamente una de las definiciones de las clases puede ser pública. Las demás clases en el archivo también se colocan en el paquete, pero no son reutilizables. Éstas se encuentran en el paquete para soportar a la clase reutilizable del archivo.
- 26.11** Los métodos que establecen los valores de datos privados deben verificar que los nuevos valores que se pretenden sean apropiados; si no lo son, los métodos establecer deben colocar las variables de instancia privadas en un estado consistente apropiado.
- 26.12** Si un método de la clase proporciona toda o parte de la funcionalidad requerida por otro método de la clase, llame a dicho método desde otro método. Esto simplifica el mantenimiento del código y reduce la probabilidad de error si la implementación del código se modifica. También es un ejemplo claro de la reutilización.
- 26.13** Acceder a los datos `private` a través de los métodos *establecer* y *obtener* no solamente protege a las variables de instancia de recibir valores no válidos, sino que además aísla a los clientes de la clase de la representación de las variables de instancia. Por lo tanto, si la representación de los datos cambia (por lo general, para reducir el almacenamiento requerido, o para mejorar el rendimiento), solamente necesita modificar las implementaciones del método; los clientes no necesitan modificación alguna mientras la interfaz proporcionada por los métodos permanezca igual.
- 26.14** Cualquier variable de una clase `static` y cualquier método de una clase `static` puede utilizarse incluso si ningún objeto de esa clase se ha instanciado.

EJERCICIOS DE AUTOEVALUACIÓN

- 26.1** Complete los espacios en blanco:
- Se accede a los miembros de una clase a través del operador _____, junto con una referencia a un objeto de la clase.
 - Se puede acceder a los miembros de una clase especificada como _____ sólo por medio de métodos de la clase.
 - Un _____ es un método especial que se utiliza para inicializar las variables de instancia de una clase.
 - Un método _____ se utiliza para asignar valores a las variables de instancia privadas de una clase.
 - Los métodos de una clase normalmente se hacen _____ y las variables de instancia de una clase normalmente se hacen _____.
 - Un método _____ se utiliza para recuperar los valores de datos privados de una clase.
 - La palabra reservada _____ introduce la definición de una clase.
 - Los miembros de una clase especificados como _____ están accesibles en cualquier parte en donde un objeto de la clase se encuentre en alcance.

- i) El operador _____ asigna de manera dinámica memoria para un objeto de un tipo especificado, y devuelve una _____ para ese tipo.
- j) Una variable de instancia _____ representa información de toda la clase.
- k) Un método declarado como **static** no puede acceder a los miembros _____ de la clase.

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

- 26.1** a) Punto (.). b) **private**. c) Constructor. d) Establecer. e) **public, private**. f) Obtener. g) **class**. h) **public**. i) **new**, referencia. j) **static**. k) No estáticos.

EJERCICIOS

- 26.2** Cree una clase llamada **Racional** para realizar operaciones aritméticas con fracciones. Escriba un programa controlador para probar su clase.

Utilice variables enteras para representar las variables de instancia privadas de la clase: el **numerador** y el **denominador**. Proporcione un método constructor que permita a un objeto de esta clase inicializarse cuando se declare. El constructor debe almacenar la fracción en forma reducida (es decir, la fracción

2/4

debe almacenarse en el objeto como 1 en el **numerador**, y 2 en el **denominador**). Proporcione un constructor sin argumentos que establezca valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos **public** para cada uno de los siguientes:

- a) Suma de dos números racionales. El resultado de la suma debe almacenarse en forma reducida.
- b) Resta de dos números racionales. El resultado de la resta debe almacenarse en forma reducida.
- c) Multiplicación de dos números racionales. El resultado de la multiplicación debe almacenarse en forma reducida.
- d) División de dos números racionales. El resultado de la división debe almacenarse en forma reducida.
- e) Impresión de números racionales en la forma **a/b**, en donde **a** es el **numerador** y **b** es el **denominador**.
- f) Impresión de números racionales en formato de punto flotante. (Considere el proporcionar capacidades de formato que permitan al usuario de la clase especificar el número de dígitos de precisión a la derecha del punto decimal.)

- 26.3** Modifique la clase **Hora2** de la figura 26.3 para que incluya el método **marcar** que incremente en un segundo la hora almacenada en un objeto **Hora2**. También proporcione un método **incrementaMinuto** para incrementar los minutos, y el método **incrementaHora** para incrementar la hora. El objeto **Hora2** siempre debe permanecer en un estado consistente. Escriba un programa controlador que pruebe el método **marcar**, el método **incrementaMinuto** y el método **incrementaHora**, para garantizar que funcionan correctamente. Asegúrese de probar los siguientes casos:
- a) Incrementar para llegar al siguiente minuto.
 - b) Incrementar para llegar a la siguiente hora,
 - c) Incrementar para llegar al día siguiente (es decir, 11:59:59 PM a 12:00:00 AM).

- 26.4** Cree una clase **Rectangulo**. La clase tiene atributos **longitud** y **ancho**, cada uno con el valor predeterminado 1. Tiene métodos que calculan el **perímetro** y el **área** del rectángulo. Tiene métodos *establecer* y *obtener*, tanto para la **longitud** como para el **ancho**. Los métodos *establecer* deben verificar que la **longitud** y el **ancho** sean números de punto flotante mayores que 0.0 y menores que 20.0.

- 26.5** Cree una clase **Rectangulo** más sofisticada que la que generó en el ejercicio anterior. Esta clase sólo almacena las coordenadas cartesianas de las cuatro esquinas del rectángulo. El constructor llama a un método *establecer* que acepta cuatro valores de coordenadas, y verifica que cada una de ellas se encuentre en el primer cuadrante y que ninguna *x* y *y* sea mayor que 20.0. El método *establecer* también verifica que las coordenadas proporcionadas, en realidad especifiquen un rectángulo. Proporcione métodos que calculen la **longitud**, el **ancho**, el **perímetro** y el **área**. La **longitud** es la más grande de las dos dimensiones. Incluya un método predicado **esCuadrado** que determine si el rectángulo es un cuadrado.

- 26.6** Modifique la clase **Rectangulo** del ejercicio anterior para que incluya un método **draw** que despliegue el rectángulo dentro de un cuadro de 25 por 25 que encierre la parte del primer cuadrante en donde se encuentra el rectángulo. Utilice métodos de la clase **Graphics** para ayudar a que se despliegue el **Rectangulo**. Si se siente ambicioso, podría incluir métodos que escalen el tamaño del rectángulo, que lo rotén y que lo muevan alrededor de la parte designada del primer cuadrante.

- 26.7** Cree una clase **EnteroEnorme** que utilice un arreglo de dígitos de 40 elementos para que almacene enteros tan grandes como 40 dígitos cada uno. Proporcione métodos **introduceEnteroEnorme**, **despliegaEnteroEnorme**, **sumaEnterosEnormes** y **restaEnterosEnormes**. Para comparar objetos de **EnteroEnorme**, proporcione métodos **esIgualQue**, **noEsIgualQue**, **esMayorQue**, **esMenorQue**, **esMayorOIgualQue** y **esMenorOIgualQue**; cada uno de éstos es un método “predicado” que simplemente devuelve **true** si las relaciones se mantienen entre los dos **EnteroEnorme**, y devuelve **false** si la relación no se mantiene. Proporcione un método predicado **esCero**. Si se siente ambicioso, también proporcione el método **multiplicaEnterosEnormes**, el método **divideEnterosEnormes** y el método **moduloDeEnterosEnormes**.
- 26.8** Cree la clase **CuentaAhorro**. Utilice una variable estática para almacenar la **tasaInteresAnual** para todas las cuentas de ahorros. Cada objeto de la clase contiene una variable de instancia privada **saldoAhorro** que indica el monto que el ahorrador tiene en depósito. Proporcione el método **calculaInteresMensual**, el cual multiplica **saldoAhorro** por **tasaInteresAnual** dividida entre 12. Este interés debe sumarse a **saldoAhorro**. Proporcione un método estático **modificaTasaInteres** que establezca un nuevo valor para **tasaInteresAnual**. Escriba un programa para probar **CuentaAhorro**. Cree dos instancias para los objetos **CuentaAhorro**, **ahorrador1** y **ahorrador2**, con saldos de \$2000.00 y \$3000.00 respectivamente. Establezca **tasaInteresAnual** en 4%, luego calcule el interés mensual e imprima los nuevos saldos para cada cuenta. Posteriormente establezca **tasaInteresAnual** en 5% y calcule el interés del siguiente mes e imprima los nuevos saldos para cada cuenta.
- 26.9** Cree la clase **EstebleceEntero**. Cada objeto de la clase puede almacenar enteros en el rango de 0 a 100. Un conjunto está representado internamente por un arreglo de valores booleanos. El elemento **a[i]** del arreglo es **true** (verdadero) si el entero *i* se encuentra en el conjunto. El elemento **a[j]** es **false** si el entero *j* no se encuentra en el conjunto. El constructor sin argumentos inicializa un conjunto llamado “conjunto vacío” (es decir, un conjunto cuya representación de arreglo contiene solamente valores **false**).
- Proporcione los siguientes métodos: el método **unionConjuntosEnteros** crea un tercer conjunto que es la unión teórica de los dos conjuntos existentes (es decir, un elemento del tercer arreglo o conjunto se establece en **true** si dicho elemento es **true** en uno o en los dos conjuntos existentes; de lo contrario, el elemento del tercer conjunto se establece en **false**). El método **interseccionConjuntosEnteros** crea un tercer conjunto que es la intersección teórica de los dos conjuntos existentes, es decir, un elemento del tercer conjunto o arreglo se establece en **false** si dicho elemento es **false** en uno o en los dos conjuntos existentes; de lo contrario el elemento del tercer conjunto se establece en **true**). El método **insertaElemento** inserta un nuevo entero **k** dentro de un conjunto (al establecer **a[k]** a **true**). El método **eliminaElemento** elimina el entero **m** (al establecer **a[m]** en **false**). El método **estableceImpresion** imprime un conjunto como una lista de números separada por espacios. Imprime solamente los elementos que están presentes en el conjunto. Imprime — para un conjunto vacío. El método **esIgualQue** determina si dos conjuntos son iguales. Escriba un programa para probar su clase **ConjuntoEnteros**. Cree varias instancias de objetos **ConjuntoEnteros**. Pruebe que todos los métodos funcionan apropiadamente.
- 26.10** Sería perfectamente razonable para la clase **Horal1** de la figura 26.1 representar la hora internamente como el número de segundos desde la medianoche, en lugar de los tres valores enteros para la hora, los minutos y los segundos. Los clientes podrían utilizar los mismos métodos públicos y obtener los mismos resultados. Modifique la clase **Horal1** de la figura 26.1 para implementar **Horal1** como el número de segundos desde medianoche y mostrar que no existe un cambio visible para los clientes de la clase.
- 26.11** (*Programa de dibujo.*) Cree un applet de dibujo que dibuje líneas, rectángulos y elipses al azar. Para este propósito, cree un conjunto de clase de formas “inteligentes” en donde los objetos de esta clases sepan cómo dibujarse a sí mismas si se les proporciona un objeto **Graphics** que les diga en dónde dibujarse (es decir, el objeto **Graphics** del applet permite a una forma dibujar en el fondo del applet). Los nombres de clases deben ser **Milinea**, **MiRecta** y **MiElipse**.
- Los datos de la clase **Milinea** deben incluir las coordenadas *x1*, *y1*, *x2* e *y2*. El método **drawLine** de la clase **Graphics** conectará mediante una línea los dos puntos proporcionados. Los datos de las clases **MiRecta** y **MiElipse** deben incluir el valor *x* de la coordenada superior izquierda, el valor *y* de la coordenada superior izquierda, un *ancho* (debe ser positivo) y una *altura* (debe ser positiva). Todos los datos de cada clase deben ser privados.
- Además de los datos, cada clase debe definir al menos los siguientes métodos públicos:
- Un constructor sin argumentos que establezca las coordenadas en 0.
 - Un constructor con argumentos que establezca las coordenadas con los valores proporcionados.
 - Métodos *establecer* para cada figura individual, que permita al programador establecer cada pieza de dato en la figura (por ejemplo, si usted tiene una variable de instancia **x1**, debe tener un método **estableceX1**).

- d) Los métodos *obtener* para cada pieza de datos individual, que permitan al programador recuperar de manera independiente cada pieza de datos de la figura (por ejemplo, si usted tiene una variable de instancia **x1**, debe tener un método **obtieneX1**).
- e) Un método **draw** con la primera línea

```
public void draw( Graphics g )
```

será llamado desde el método **paint** del applet para dibujar una figura en la pantalla.

Los métodos anteriores son indispensables. Si usted desea proporcionar más métodos para mayor flexibilidad, hágalo.

Comience con la definición de la clase **MiLinea** y un applet para probar sus clases. El applet debe tener una variable de instancia línea de **MiLinea** que pueda hacer referencia a un objeto **MiLinea** (creado en el método **init** del applet con coordenadas al azar). El método **paint** del applet debe dibujar la figura con una instrucción como

```
linea.draw( g );
```

en donde **linea** es una referencia a **MiLinea** y **g** es el objeto de **Graphics** que la forma utilizará para dibujarse a sí misma en el applet.

Después, modifique la referencia individual a **MiLinea** dentro de un arreglo de referencias a **MiLinea** y copie el código para varios objetos **MiLinea** dentro del programa de dibujo. El método **paint** del applet debe recorrer el arreglo de objetos **MiLinea** y dibujar cada uno.

Una vez que la parte anterior ya funcione, debe definir las clases **MiElipse** y **MiRecta**, y agregar los objetos de estas clases a los arreglos **MiRecta** y **MiElipse**. El método **paint** del **applet** debe recorrer cada arreglo y dibujar cada figura. Cree cinco figuras de cada tipo.

Una vez que el applet funcione, seleccione **Volver a cargar** del menú **Subprograma** para volver a cargar el applet. Esto provocará que el applet elija nuevos números al azar para dibujar las figuras.

En el capítulo 27, modificaremos este ejercicio para aprovechar las similitudes entre las clases, y así evitar el reinventar la rueda.

Programación orientada a objetos en Java

Objetivos

- Comprender la herencia y la reutilización de software.
- Comprender las superclases y las subclases.
- Apreciar cómo es que el polimorfismo hace que los sistemas sean extensibles y que se puedan mantener.
- Comprender la diferencia entre clases abstractas y clases concretas.
- Aprender cómo crear clases abstractas (**abstract**) e interfaces.

No digas que conoces completamente a alguien, hasta que hayas compartido una herencia con él.

Johann Kaspar Lavater



Este método es para definir como el número de una clase a la clase de todas las clases similares a la clase dada.

Bertrand Russell

Es bueno heredar una biblioteca, pero es mejor formar una.

Augustine Birrell

Las proposiciones generales no deciden casos concretos.

Oliver Wendell Holmes

*Un filósofo de imponente estatura no piensa en un vacío.
Incluso sus ideas más abstractas son, hasta cierto punto,
condicionadas por lo que se sabe, o no se sabe, en la época
en la que vive.*

Alfred North Whitehead

Plan general

- 27.1 Introducción
- 27.2 Superclases y subclases
- 27.3 Miembros `protected`
- 27.4 Relación entre objetos de superclases y objetos de subclases
- 27.5 Conversión implícita de un objeto de una subclase en un objeto de una superclase
- 27.6 Ingeniería de software con herencia
- 27.7 Composición *versus* herencia
- 27.8 Introducción al polimorfismo
- 27.9 Campos de tipo e instrucciones `switch`
- 27.10 Método de vinculación dinámica
- 27.11 Métodos y clases `final`
- 27.12 Superclases abstractas y clases concretas
- 27.13 Ejemplo de polimorfismo
- 27.14 Nuevas clases y vinculación dinámica
- 27.15 Ejemplo práctico: Herencia de interfaz y de implementación
- 27.16 Ejemplo práctico: Creación y uso de interfaces
- 27.17 Definiciones de clases internas
- 27.18 Notas sobre las definiciones de clases internas
- 27.19 Clases envolventes para tipos primitivos

Resumen • Terminología • Errores comunes de programación • Tips para prevenir errores • Tips de rendimiento
• Observaciones de ingeniería de software • Ejercicios de autoevaluación • Resuestas a los ejercicios de autoevaluación • Ejercicios

27.1 Introducción

En este capítulo explicamos la programación orientada a objetos (POO) y sus tecnologías componentes clave: la *herencia* y el *polimorfismo*. La herencia es una forma de reutilización de software, en la que se crean nuevas clases a partir de clases existentes, absorbiendo sus atributos y sus comportamientos y mejorándolas con capacidades que requieren las nuevas clases. La reutilización de software ahorra tiempo en el desarrollo de programas, lo cual motiva el uso de software de alta calidad probado y depurado, con lo que se reducen los problemas que se generan cuando un sistema empieza a utilizarse. Éstas son posibilidades excitantes. El polimorfismo nos permite escribir programas de modo general para manejar una amplia variedad de clases relacionadas existentes. El polimorfismo facilita el agregar nuevas capacidades a un sistema. La herencia y el polimorfismo son técnicas efectivas para lidiar con la complejidad del software.

Cuando el programador crea una nueva clase, en lugar de escribir variables y métodos de instancia completamente nuevos, puede designar que la nueva clase herede las variables y los métodos de instancia de una *superclase* previamente definida. A la nueva clase se le conoce como una *subclase*. Cada subclase por sí misma se vuelve una candidata para ser una superclase para algunas subclases futuras.

La *superclase directa* de una subclase es la superclase de la que la subclase directamente hereda (vía la palabra reservada `extends`). Una superclase indirecta hereda desde dos o más niveles superiores en la jerarquía de clase.

Por medio de la *herencia simple*, una clase se deriva de una superclase. Java no soporta la *herencia múltiple* (como C++ lo hace), pero sí soporta la idea de las *interfaces*. Las interfaces ayudan a Java a tener muchas de las ventajas de la herencia múltiple sin los problemas asociados. En este capítulo explicaremos los detalles de las interfaces; consideraremos los principios generales y un ejemplo detallado sobre la creación y el uso de las interfaces.

Una subclase normalmente agrega por su cuenta variables y métodos de instancia, por lo que una subclase generalmente es más grande que su superclase. Una subclase es más específica que su superclase y representa un grupo más pequeño de objetos. Con la herencia simple, la subclase inicia prácticamente igual que la superclase. La fortaleza real de la herencia proviene de la habilidad de definir en la subclase agregados, o reemplazos, para las características heredadas de la superclase.

Todo objeto de una subclase es también un objeto de la superclase de esa subclase. Sin embargo, lo inverso no es verdad; los objetos de una superclase no son objetos de las subclases de esa superclase. Nosotros aprovecharemos la relación “el objeto de una subclase es un objeto de la superclase” para realizar algunas manipulaciones poderosas. Por ejemplo, por medio de la herencia podemos vincular una amplia variedad de objetos diferentes, relacionados con una superclase común, en una lista ligada de objetos de una superclase. Esto permite que una variedad de objetos se procesen en una manera general. Como veremos en este capítulo, es la idea central de la programación orientada a objetos.

En este capítulo agregamos una nueva forma de control de acceso a miembros, a saber, el acceso **protected** (protegido). Los métodos de una subclase y los métodos de otras clases en el mismo paquete de la superclase pueden acceder a los miembros **protected** de la superclase.

La experiencia en construir sistemas de software indica que partes importantes de código lidian con casos especiales muy relacionados. En tales sistemas se torna difícil ver la “imagen completa”, ya que el diseñador y el programador se preocupan por los casos especiales. La programación orientada a objetos proporciona diversas formas para “ver el bosque a través de los árboles”; un proceso llamado *abstracción*.

Si un programa por procedimientos tiene muchos casos especiales muy relacionados, entonces es común ver estructuras **switch** o estructuras **if/else** anidadas que diferencian los casos especiales y proporcionan la lógica de procesamiento para manejar individualmente cada caso. Mostraremos cómo utilizar la herencia y el polimorfismo para remplazar dicha lógica de **switch** con una lógica mucho más sencilla.

Plantearemos la diferencia entre la *relación es un* y la *relación tiene un*. *Es un* es herencia. En una relación *es un*, un objeto de un tipo correspondiente a una subclase también puede tratarse como un objeto de un tipo de su superclase. *Tiene un* es composición (como explicamos en el capítulo 26). En una relación *tiene un*, un objeto de una clase tiene como miembros a uno o más objetos de otras clases. Por ejemplo, un automóvil *tiene un* volante.

Los métodos de una subclase podrían necesitar acceder a ciertas variables y ciertos métodos de instancia de su superclase.

Observación de ingeniería de software 27.1



Una subclase no puede acceder directamente a miembros **private** de su superclase.

Éste es un aspecto crucial de la ingeniería de software en Java. Si una subclase pudiera acceder a los miembros **private** de una superclase, se violaría el ocultamiento de información en la superclase.

Tip para prevenir errores 27.1



Ocultar los miembros **private** es una gran ayuda al probar, depurar y modificar correctamente los sistemas. Si una subclase pudiera acceder a los miembros **private** de su superclase, entonces sería posible que las clases derivadas de esa subclase accedieran también a esos datos, y así sucesivamente. Esto propagaría el acceso a lo que se supone deberían ser datos **private**, y los beneficios del ocultamiento de información se perderían a lo largo de la jerarquía de la clase.

Una subclase en el mismo paquete de su superclase puede acceder a los miembros **public**, **protected** y miembros de acceso al paquete de su superclase. Los miembros de una superclase que no deben acceder a una subclase por medio de la herencia, se declaran como **private** en la superclase. Una subclase puede efectuar modificaciones de estado a los miembros **private** de una superclase, sólo a través de métodos **public**, **protected** y de acceso a paquetes provistos en la superclase y heredados a la subclase.

Un problema con la herencia es que una subclase puede heredar métodos que no necesita, o que no debe tener. Cuando un miembro de una superclase es inadecuado para una subclase, ese miembro puede *redefinirse* en la subclase con una implementación adecuada.

Tal vez lo más excitante sea la noción de que las nuevas clases pueden ser herederas de muchas *bibliotecas de clases*. Las empresas desarrollan sus propias bibliotecas de clases y aprovechan otras disponibles alrededor del mundo. Algun día, la mayoría del software se construirá a partir de *componentes reutilizables estandarizados*,

tal como se construye actualmente la mayoría del hardware. Esto ayudará a cumplir con el reto de desarrollar software poderoso que necesitaremos en el futuro.

27.2 Superclases y subclases

Con frecuencia, un objeto de una clase también *es un* objeto de otra clase. Un rectángulo ciertamente *es un* cuadrilátero (como los cuadrados, los paralelogramos y los trapezoides). Entonces, puede decirse que la clase **Rectangulo** hereda de la clase **Cuadrilatero**. En este contexto, la clase **Cuadrilatero** es una superclase y la clase **Rectangulo** es una subclase. Un rectángulo *es un* tipo específico de cuadrilátero, pero es incorrecto afirmar que un cuadrilátero *es un* rectángulo (el cuadrilátero podría ser un paralelogramo). La figura 27.1 muestra diversos ejemplos de herencia simple de superclases y subclases potenciales.

La herencia normalmente produce subclases con *más* características que sus superclases, por lo que los términos superclase y subclase pueden ser confusos. Sin embargo, existe otra manera de ver estos términos, la cual hace clara la relación. Todo objeto de una subclase *es un* objeto de su superclase, y una superclase puede tener muchas subclases, por lo que el conjunto de objetos representados por una superclase normalmente es más grande que el conjunto de objetos representados por cualquier subclase de esa superclase. Por ejemplo, la superclase **Vehiculo** representa de manera general a todos los vehículos, como automóviles, camiones, botas, bicicletas, etcétera. Sin embargo, la subclase **Automovil** representa sólo a un pequeño subconjunto de todos los vehículos en el mundo.

Las relaciones de herencia forman estructuras jerárquicas parecidas a un árbol. Una superclase existe en una relación jerárquica con sus subclases. Ciertamente, una clase puede existir por sí misma, pero es cuando una clase se utiliza con el mecanismo de la herencia, que la clase se vuelve una superclase que proporciona atributos y comportamientos a otras clases, o que la clase se vuelve una subclase que hereda dichos atributos y comportamientos.

Desarrollemos una jerarquía de herencia simple para figuras. Los círculos, cuadrados, cubos y tetraedros son diferentes tipos de figuras. Algunas de estas figuras pueden dibujarse en dos dimensiones, y algunas otras deben modelarse en tres dimensiones. Esto arroja la jerarquía de herencia que aparece en la figura 27.2. Observe que esta jerarquía podría contener muchas otras clases. Por ejemplo, los cuadrados y los rectángulos son cuadriláteros. Las flechas en la jerarquía representan la relación *es un*. Por ejemplo, basándonos en esta jerarquía de clase podemos decir que “un **Cuadrado** *es una FiguraBidimensional*”, o que “un **Cubo** *es una FiguraTridimensional*”. **Figura** es la *superclase directa* tanto de **FiguraBidimensional** como de **FiguraTridimensional**. **Figura** es una *superclase indirecta* de todas las demás clases del diagrama de jerarquía.

Superclase	Subclases
Estudiante	EstudianteTitulado EstudianteUniversitario
Figura	Circulo Triangulo Rectangulo
Prestamo	PrestamoAutomotriz PrestamoMejorarCasa PrestamoHipotecario
Empleado	EmpleadoDocente EmpleadoAdministrativo
Cuenta	CuentaCheques CuentaAhorros

Figura 27.1 Algunos ejemplos de herencia simple.

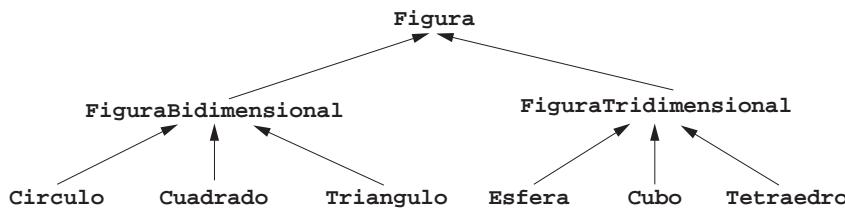


Figura 27.2 Una parte de la jerarquía de la clase **Figura**.

Además, si partimos de la parte inferior del diagrama, podemos seguir las flechas y aplicar la relación *es un* hacia arriba, hasta llegar a la parte superior de la jerarquía de la superclase. Por ejemplo, un **Tetraedro** *es una FiguraTridimensional* y también *es una Figura*. En Java, un **Tetraedro** también es un **Object**, ya que todas las clases en Java tienen a **Object** como una de sus superclases directas o indirectas. Por lo tanto, todas las clases en Java tienen una relación jerárquica en la que comparten los 11 métodos definidos por la clase **Object**, la cual incluye los métodos **toString** y **finalize** que explicamos anteriormente. Explicaremos otros métodos de la clase **Object** conforme los necesitemos en el texto.

En el mundo existen muchos ejemplos de jerarquías, pero los estudiantes no están acostumbrados a categorizar al mundo de esta manera, por lo que son necesarios ciertos ajustes a su pensamiento. De hecho, los estudiantes de biología han tenido prácticas con jerarquías. Todo lo que estudiamos en biología está agrupado en una jerarquía encabezada por los seres vivos, los cuales pueden ser plantas, animales, etcétera.

Para especificar que la clase **FiguraBidimensional** se deriva (o hereda de) la clase **Figura**, la clase **FiguraBidimensional** podría definirse en Java como:

```
class FiguraBidimensional extends Figura { ... }
```

Con la herencia, los miembros **private** de una superclase no son directamente accesibles para las subclases de esa clase. Los miembros de acceso al paquete de la superclase sólo son accesibles en una subclase, si la superclase y su subclase están en el mismo paquete. Todos los demás miembros de la superclase se vuelven miembros de la subclase, utilizando su acceso a miembros original (es decir, los miembros **public** de la superclase se vuelven miembros **public** de la subclase, y los miembros **protected** de la superclase se vuelven miembros **protected** de la subclase).



Observación de ingeniería de software 27.2

Los constructores nunca se heredan; éstos son específicos de la clase en la que están definidos.

Es posible tratar a los objetos de una superclase y a los objetos de una subclase de manera similar; esa similitud se expresa en los atributos y comportamientos de la superclase. Los objetos de todas las clases derivadas de una superclase común pueden tratarse como objetos de esa superclase.

Consideraremos muchos ejemplos en los que aprovecharemos esta relación con una programación sencilla no disponible en lenguajes no orientados a objetos como C.

27.3 Miembros **protected**

Los miembros **public** de una superclase son accesibles desde cualquier parte del programa que tenga una referencia hacia ese tipo de superclase o hacia uno de los tipos de su subclase. Los miembros **private** de una superclase sólo son accesibles en los métodos de esa superclase.

Los miembros **protected** de una superclase sirven como un nivel intermedio de protección entre el acceso **public** y el **private**. Se puede acceder a los miembros **protected** de una superclase sólo por medio de métodos de la superclase, por medio de métodos de subclases y por medio de métodos de otras clases en el mismo paquete (los miembros **protected** tienen acceso a paquetes).

Los métodos de subclases normalmente pueden hacer referencia a miembros **public** y **protected** de la superclase, simplemente utilizando los nombres de los miembros. Cuando un método de una subclase *redefine* un método de una superclase (explicado en la sección 27.4), se puede acceder al método de la superclase

desde la subclase, precediendo el nombre del método de la superclase con la palabra reservada **super**, seguida por el operador punto (.). Ilustramos esta técnica varias veces a lo largo del capítulo.

27.4 Relación entre objetos de superclases y objetos de subclases

Un objeto de una subclase puede tratarse como un objeto de su superclase. Esto hace posible realizar algunas manipulaciones interesantes. Por ejemplo, a pesar del hecho de que los objetos de una variedad de clases derivadas de una superclase en particular pueden ser muy diferentes entre sí, podemos crear un arreglo de referencias hacia ellos; siempre y cuando los tratemos como objetos de una superclase. Sin embargo, lo contrario no es verdad: un objeto de una superclase no es automáticamente un objeto de una subclase.

Error común de programación 27.1



Tratar a un objeto de una superclase como un objeto de una subclase puede ocasionar errores.

Sin embargo, se puede utilizar una conversión de tipo explícita para convertir una referencia de una superclase en una referencia de una subclase. Esto únicamente puede hacerse cuando la referencia de la superclase en realidad hace referencia a un objeto de una subclase; de lo contrario, Java indica una **ClassCastException**; una indicación de que la operación de conversión de tipo no está permitida.

Error común de programación 27.2



Asignar un objeto de una superclase a una referencia de una subclase (sin una conversión de tipo), es un error de sintaxis.

Observación de ingeniería de software 27.3



Si un objeto se ha asignado a una referencia de una de sus superclases, es aceptable convertir el tipo de ese objeto de regreso a su propio tipo. De hecho, esto debe hacerse para enviar a ese objeto cualquiera de los mensajes que no aparecen en esa superclase.

Nuestro primer ejemplo de herencia aparece en la figura 27.3. Todos los applets que definimos han utilizado alguna de las técnicas que presentamos aquí. Ahora formalizaremos el concepto de la herencia.

```

1 // Figura 27.3: Punto.java
2 // Definición de la clase Punto
3
4 public class Punto {
5     protected int x, y; // coordenadas del Punto
6
7     // Constructor sin argumentos
8     public Punto()
9     {
10         // llamada implícita al constructor de la superclase
11         establecePunto( 0, 0 );
12     } // fin del constructor Punto
13
14     // Constructor
15     public Punto( int a, int b )
16     {
17         // llamada implícita al constructor de la superclase
18         establecePunto( a, b );
19     } // fin del constructor Punto
20
21     // Establece las coordenadas x y y del Punto
22     public void establecePunto( int a, int b )
23     {

```

Figura 27.3 Asignación de referencias de subclases a referencias de superclases; **Punto.java**.
(Parte 1 de 2.)

```

24     x = a;
25     y = b;
26 } // fin del método establecePunto
27
28 // obtiene la coordenada x
29 public int obtieneX() { return x; }
30
31 // obtiene la coordenada y
32 public int obtieneY() { return y; }
33
34 // convierte el punto a una representación como String representation
35 public String toString()
36     { return "[" + x + ", " + y + "]"; }
37 } // fin de la clase Punto

```

Figura 27.3 Asignación de referencias de subclases a referencias de superclases; **Punto.java**. (Parte 2 de 2.)

```

38 // Figura 27.3: Circulo.java
39 // Definición de la clase Circulo
40
41 public class Circulo extends Punto { // hereda desde punto
42     protected double radio;
43
44     // Constructor sin argumentos
45     public Circulo()
46     {
47         // llamada implícita al constructor de la superclase
48         estableceRadio( 0 );
49     } // fin del constructor Circulo
50
51     // Constructor
52     public Circulo( double r, int a, int b )
53     {
54         super( a, b ); // llama al constructor de la superclase
55         estableceRadio( r );
56     } // fin del constructor Circulo
57
58     // Establece el radio de Circulo
59     public void estableceRadio( double r )
60     { radio = ( r >= 0.0 ? r : 0.0 ); }
61
62     // Obtiene el radio de Circulo
63     public double obtieneRadio() { return radio; }
64
65     // Calcula el área de Circulo
66     public double area() { return Math.PI * radio * radio; }
67
68     // convierte el Circulo a String
69     public String toString()
70     {
71         return "Centro = " + "[" + x + ", " + y + "]"
72             + "; Radio = " + radio;
73     } // fin del método toString
74 } // fin de la clase Circulo

```

Figura 27.3 Asignación de referencias de subclases a referencias de superclases; **Circulo.java**.

```
75 // Figura 27.3: PruebaHerencia.java
76 // Demostración de la relación "es un"
77 import java.text.DecimalFormat;
78 import javax.swing.JOptionPane;
79
80 public class PruebaHerencia {
81     public static void main( String args[] )
82     {
83         Punto refPunto, p;
84         Circulo refCirculo, c;
85         String salida;
86
87         p = new Punto( 30, 50 );
88         c = new Circulo( 2.7, 120, 89 );
89
90         salida = "Punto p: " + p.toString() +
91                 "\nCirculo c: " + c.toString();
92
93         // utiliza la relación "es un" para hacer referencia a Circulo
94         // mediante una referencia a Punto
95         refPunto = c;    // asigna Circulo a refPunto
96
97         salida += "\n\nCirculo c (via refPunto): " +
98                 refPunto.toString();
99
100        // Utiliza la conversión hacia abajo (convierte una referencia a una
101        // superclase a un tipo de dato subclase) para asignar refPunto a
102        // refCirculo
103        refCirculo = (Circulo) refPunto;
104
105        salida += "\n\nCirculo c (mediante refCirculo): " +
106                 refCirculo.toString();
107
108        DecimalFormat precision2 = new DecimalFormat( "0.00" );
109        salida += "\nÁrea de c (via refCirculo): " +
110                 precision2.format( refCirculo.area() );
111
112        // Intenta hacer referencia a un objeto Punto
113        // mediante la referencia a Circulo
114        if ( p instanceof Circulo ) {
115            refCirculo = (Circulo) p; // línea 40 en Prueba.java
116            salida += "\n\nconversión exitosa";
117        }
118        else
119            salida += "\n\np no hace referencia a Circulo";
120
121        JOptionPane.showMessageDialog( null, salida,
122             "Demuestra la \"relación \" es un",
123             JOptionPane.INFORMATION_MESSAGE );
124
125        System.exit( 0 );
126    } // fin de main
127 } // fin de la clase PruebaHerencia
```

Figura 27.3 Asignación de referencias de subclases a referencias de superclases;
PruebaHerencia.java.(Parte 1 de 2.)

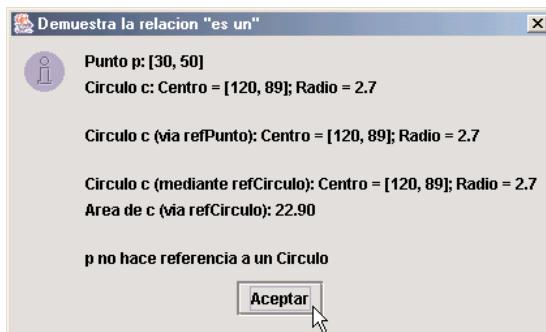


Figura 27.3 Asignación de referencias de subclases a referencias de superclases; **PruebaHerencia.java**. (Parte 2 de 2.)

En Java, toda definición de clase debe extender a otra clase. Sin embargo, observe que la clase **Punto** (línea 4) no utiliza explícitamente la palabra reservada **extends**. Si la definición de una nueva clase no extiende explícitamente una definición de clase existente, Java utiliza implícitamente la clase **Object** (paquete **java.lang**) como la superclase para la definición de la nueva clase. La clase **Object** proporciona un conjunto de métodos que pueden utilizarse con cualquier objeto de cualquier clase.



Observación de ingeniería de software 27.4

Toda clase en Java extiende a **Object**, a menos que se especifique lo contrario en la primera línea de la definición de la clase. Por lo tanto, la clase **Object** es la superclase de toda la jerarquía de clases de Java.

Las líneas 1 a 37 muestran la definición de la clase **Punto**. Las líneas 38 a 74 muestran la definición de la clase **Círculo**; veremos que la clase **Círculo** hereda de la clase **Punto**. Las líneas 75 a 126 muestran una aplicación que demuestra la asignación de referencias de subclase a referencias de superclase, y la conversión de tipo de referencias de superclase a referencias de subclase.

Primero examinemos la definición de la clase **Punto** de las líneas 1 a 37. Los servicios **public** de la clase **Punto** incluye los métodos **establecePunto**, **obtieneX**, **obtieneY**, **toString** y dos constructores **Punto**. Las variables de instancia **x** y **y** de **Punto** se especifican como **protected**. Esto evita que los clientes de objetos **Punto** accedan directamente a los datos (a menos que sean clases del mismo paquete), pero permite a las clases derivadas de **Punto** acceder directamente a las variables de instancia heredadas. Si los datos se especificaran como **private**, los métodos no privados de **Punto** tendrían que utilizarse para acceder a los datos, incluso las subclases. Observe que el método **toString** de **Punto** redefine el método **toString** original de la clase **Object**.

Los constructores de la clase **Punto** (líneas 8 y 15) deben llamar al constructor de la clase **Object**. De hecho, todo constructor de subclases es necesario para llamar al constructor de su superclase directa, ya sea implícita o explícitamente, como su primera tarea (por el momento, la sintaxis de esta llamada se explica con la clase **Círculo**). Si no hay una llamada explícita al constructor de la superclase, Java intenta llamar al constructor predeterminado de la superclase. Observe que las líneas 10 y 17 son comentarios que indican en dónde ocurre la llamada al constructor predeterminado de la superclase **Object**.

La clase **Círculo** (líneas 38 a 74) hereda de la clase **Punto**. Esto se especifica en la primera línea de la definición de la clase

```
public class Circulo extends Punto { // hereda de Punto
```

La palabra reservada **extends** en la definición de la clase indica la herencia. Todos los miembros (no privados) de la clase **Punto** (excepto los constructores) se heredan a la clase **Círculo**. Por lo tanto, la interfaz **public** de **Círculo** incluye los métodos **public** de **Punto**, así como los dos constructores sobrecargados de **Círculo** y los métodos de **Círculo** **estableceRadio**, **obtieneRadio**, **area** y **toString**. Observe que el método **area** (línea 66) utiliza la constante predefinida **Math.PI** de la clase **Math** (paquete **java.lang**) para calcular el área de un círculo.

Los constructores de **Círculo** (líneas 45 y 52) deben invocar un constructor **Punto** para inicializar la parte de superclase (variables **x** y **y** heredadas de **Punto**) de un objeto **Círculo**. El constructor predeterminado de la línea 45 no llama explícitamente a un constructor **Punto**, por lo que Java llama al constructor predeterminado de **Punto** (definido en la línea 8), el cual inicializa en ceros a los miembros de la superclase **x** y **y**. Si la clase **Punto** contuviera sólo el constructor de la línea 15 (es decir, no proporcionara un constructor predeterminado), ocurriría un error de compilación.

La línea 54 del cuerpo del segundo constructor **Círculo**

```
super( a, b ); // llamada explícita al constructor de la superclase
```

invoca explícitamente al constructor **Punto** (definido en la línea 11) por medio de la sintaxis de llamada al constructor de la superclase [es decir, la palabra reservada **super**, seguida por un conjunto de paréntesis que contienen los argumentos del constructor de la superclase (en este caso los valores **a** y **b** son pasados para inicializar a los miembros de la superclase **x** y **y**)]. La llamada al constructor de la superclase debe ser la primera línea del cuerpo del constructor de la subclase. Para llamar explícitamente al constructor predeterminado de la superclase, utilice la instrucción

```
super(); // llamada explícita al constructor predeterminado de la superclase
```

Error común de programación 27.3



*Si una subclase hace una llamada **super** al constructor de su superclase, y esta llamada no es la primera instrucción en el constructor de la subclase, es un error de sintaxis.*

Error común de programación 27.4



*Si los argumentos de una llamada **super** de una subclase al constructor de su superclase no coinciden con los parámetros especificados en una de las definiciones del constructor de la superclase, es un error de sintaxis.*

Una subclase puede redefinir el método de una superclase, utilizando la misma firma; a esto se le conoce como *redefinir* un método de superclase. Siempre que se menciona a un método por su nombre en la subclase, se llama a la versión de la subclase. De hecho, hemos redefinido métodos en todos los applets del libro. Cuando extendemos **JApplet** para crear una nueva clase applet, la nueva clase hereda las versiones de **init** y **paint** (y muchos otros métodos). Cada vez que definimos **init** o **paint**, redefinimos la versión original que se heredó. Además, cuando proporcionamos el método **toString** para las clases del capítulo 26, redefinimos la versión original de **toString** provista por la clase **Object**. Como veremos pronto, la referencia **super**, seguida por el operador punto, puede utilizarse para acceder a la versión original de la superclase de ese método desde la subclase.

Observe que el método **toString** de la clase **Círculo** (línea 69) redefine el método **toString** de la clase **Punto** (línea 35). El método **toString** de la clase **Punto** redefine el método original **toString** provisto por la clase **Object**. La clase **Object** proporciona el método original **toString**, por lo que todas las clases heredan un método **toString**. Este método se utiliza para convertir cualquier objeto de cualquier clase en una representación **String** y algunas veces es llamado implícitamente por el programa (por ejemplo, cuando se agrega un objeto a una **String**). El método **toString** de **Círculo** accede directamente a las variables de instancia **protected x** y **y** que se heredaron de la clase **Punto**. Los valores **x** y **y** se utilizan como parte de la representación **String** de **Círculo**. De hecho, si estudia el método **toString** de **Punto** y el método **toString** de la clase **Círculo**, notará que **toString** de **Círculo** utiliza exactamente el mismo formato que **toString** de **Punto** para las partes **Punto** del **Círculo**. Para llamar a **toString** de **Punto** desde la clase **Círculo**, utilice la expresión

```
super.toString()
```

Observación de ingeniería de software 27.5



Una redefinición de un método de una superclase en una subclase no tiene la misma firma que el método de la superclase. Tal redefinición no es la redefinición de un método, sino un simple ejemplo de la sobrecarga de métodos.

Observación de ingeniería de software 27.6



*Cualquier objeto puede convertirse en una **String** con una llamada explícita o implícita al método **toString** del objeto.*

Observación de ingeniería de software 27.7



Toda clase debe redefinir el método `toString` para devolver información útil sobre los objetos de esa clase.

Error común de programación 27.5



Si un método de una superclase y un método en su subclase tienen la misma firma pero diferente tipo de retorno, es un error de sintaxis.

La aplicación (líneas 75 a 126) crea las instancias del objeto `p` de `Punto` y del objeto `c` de `Circulo` en las líneas 87 y 88 de `main`. Las representaciones `String` de cada objeto se adjuntan a `String salida` para mostrar que se inicializaron correctamente (líneas 90 y 91). Vea las dos primeras líneas de la salida de la captura de pantalla para confirmar esto.

La línea 95

```
refPunto = c; // asigna Circulo a refPunto
```

asigna a `Circulo c` (una referencia hacia un objeto de subclase) a `refPunto` (una referencia de superclase). En Java, siempre es aceptable asignar una referencia de subclase a una referencia de superclase (debido a la relación de herencia *es un*). Un `Circulo` *es un Punto*, ya que la clase `Circulo` extiende a la clase `Punto`. Como veremos, asignar una referencia de superclase a una referencia de subclase es peligroso.

Las líneas 97 y 98 agregan el resultado de `refPunto.toString()` a la `String salida`. De manera interesante, cuando a esta `refPunto` se le envía al mensaje de `toString`, Java sabe que el objeto realmente es un `Circulo`, por lo que elige el método `toString` de `Circulo`, en lugar de usar el método `toString` de `Punto`, como pudo haber esperado. Éste es un ejemplo de *polimorfismo* y de *vinculación dinámica*, conceptos que trataremos con detalle más adelante en este capítulo. El compilador ve la expresión anterior y hace la pregunta “¿el tipo de dato de la referencia `refPunto` (es decir, `Punto`) tiene un método `toString` sin argumentos?” La respuesta a esta pregunta es sí (vea la definición de `toString` de `Punto` en la línea 35). El compilador simplemente verifica la sintaxis de la expresión y se asegura de que el método existe. En tiempo de ejecución, el intérprete hace la pregunta “¿de qué tipo es el objeto al que `refPunto` hace referencia?”. Todo objeto en Java sabe su propio tipo de dato, por lo que la respuesta a esta pregunta es que `refPunto` hace referencia a un objeto `Circulo`. Basándose en esta respuesta, el intérprete llama al método `toString` del tipo de dato del objeto (es decir, el método `toString` de la clase `Circulo`). Vea la tercera línea de la salida para confirmar esto. Las dos principales técnicas que utilizamos para lograr este efecto son: 1) extender la clase `Punto` para crear la clase `Circulo`, y 2) redefinir el método `toString` con exactamente la misma firma en la clase `Punto` y en la clase `Circulo`.

La línea 102

```
refCirculo = (Circulo) refPunto;
```

convierte el tipo de `refPunto` (la cual admite hacer referencia a `Circulo` en este punto de la ejecución del programa) en un `Circulo`, y asigna el resultado a `refCirculo` (esta conversión de tipo sería peligrosa si `refPunto` realmente hiciera referencia a `Punto`, como explicaremos pronto). Después utilizamos `refCirculo` para agregar a `String salida` los diferentes hechos sobre la `refCirculo` de `Circulo`. Las líneas 104 y 105 invocan al método `toString` para agregar la representación `String` de `Circulo`. Las líneas 107 a 109 agregan el `area` del `Circulo` con el formato de una instancia de la clase `DecimalFormat` (paquete `java.text`) llamada `precision2` que da formato al número con dos dígitos a la derecha del punto decimal. El formato “`0.00`” (especificado en la línea 107) utiliza el `0` dos veces para indicar el número adecuado de dígitos después del punto decimal. Cada `0` es un lugar decimal requerido. El `0` a la izquierda del punto decimal indica un mínimo de un dígito a la izquierda del punto decimal.

Después, la estructura `if/else` de las líneas 113 a 118 intenta una conversión de tipo peligrosa en la línea 114. Convertimos el tipo de `p` de `Punto` en un `Circulo`. Si esto se intenta en tiempo de ejecución, Java determinaría que `p` realmente hace referencia a `Punto`, reconocería la conversión de tipo a `Circulo` como peligrosa, e indicaría una conversión inadecuada con el mensaje de `ClassCastException`. Sin embargo, evitamos que esta instrucción se ejecute con la condición `if`

```
if( p instanceof Circulo ) {
```

la cual utiliza el operador `instanceof` para determinar si el objeto al que `p` se refiere es un `Círculo`. Esta condición da como resultado `true` sólo si el objeto al que `p` se refiere es un `Círculo`; de lo contrario resulta en `false`. La referencia `p` no se refiere a un `Círculo`, por lo que la condición falla y se agrega una `String` a `salida`, la cual indica que `p` no se refiere a un `Círculo`.

Si eliminamos la prueba `if` del programa y lo ejecutamos, se genera el siguiente mensaje en tiempo de ejecución:

```
Exception in thread "main"
java.lang.ClassCastException: Punto
at PruebaHerencia.main(PruebaHerencia.java:40)
```

Tales mensajes de error normalmente incluyen el nombre del archivo (`PruebaHerencia.java`) y el número de línea en la que ocurrió el error, para que pueda ir a esa línea específica del programa para depurarla. Observe que el número de línea especificado (`PruebaHerencia.java:40`) es diferente de los números de línea para el archivo `PruebaHerencia.java` que aparece en el texto. Esto se debe a que los ejemplos del texto están numerados consecutivamente para todos los archivos del mismo programa, con propósitos explicativos. Si abre el archivo `PruebaHerencia.java` en un editor, descubrirá que el error realmente ocurrió en la línea 40 (la cual es la línea 114 del programa completo).

27.5 Conversión implícita de un objeto de una subclase en un objeto de una superclass

A pesar del hecho de que un objeto de una subclase es un objeto de una superclass, el tipo de la subclase y el tipo de la superclass son diferentes. Los objetos de una subclase pueden tratarse como objetos de la superclass. Esto tiene sentido debido a que la subclase tiene miembros que corresponden a cada uno de los miembros de la superclass; recuerde que la subclase normalmente tiene más miembros que la superclass. La asignación en la otra dirección no está permitida, ya que asignar un objeto de una superclass a una referencia de una subclase dejaría indefinidos a los miembros adicionales de la subclase.

Una referencia a un objeto de una subclase se convertiría implícitamente en una referencia a un objeto de la superclass, ya que el objeto de la subclase es un objeto de la superclass a través de la herencia.

Existen cuatro posibles formas de mezclar y de hacer coincidir referencias de superclases y referencias de subclases con objetos de superclases y objetos de subclases:

1. Hacer referencia a un objeto de una superclass con una referencia de una superclass es directo.
2. Hacer referencia a un objeto de una subclase con una referencia de una subclase es directo.
3. Hacer referencia a un objeto de una subclase con una referencia de una superclass es seguro, ya que el objeto de una subclase también es un objeto de su superclass. Tal código sólo puede hacer referencia a miembros de la superclass. Si este código hace referencia sólo a miembros de la subclase, a través de referencias de superclass, el compilador reportará un error de sintaxis.
4. Hacer referencia a un objeto de una superclass con una referencia de subclase es un error de sintaxis. La referencia de subclase primero debe convertirse al tipo de una referencia de superclass.

Error común de programación 27.6



Asignar un objeto de subclase a una referencia de superclass, y después intentar hacer referencia sólo a miembros de la subclase con la referencia de superclass, es un error de sintaxis.

Aparentemente es conveniente tratar a los objetos de subclases como objetos de superclases, y hacer esto manipulando todos estos objetos con referencias de superclases aparentemente también es un problema. Por ejemplo, en un sistema de nómina nos gustaría recorrer un arreglo de empleados y calcular el pago semanal para cada persona. Sin embargo, la intuición nos dice que utilizar referencias de superclases permitiría al programa llamar únicamente a la rutina de superclass que calcula la nómina (si en realidad hay tal rutina en la superclass). Nosotros necesitamos una manera de invocar la rutina adecuada que calcule la nómina para cada objeto, ya sea un objeto de superclass o un objeto de subclase, y hacer esto simplemente utilizando la referencia de superclass. De hecho, es precisamente así como se comporta Java, y lo explicamos en este capítulo cuando consideramos el polimorfismo y la vinculación dinámica.

27.6 Ingeniería de software con herencia

Podemos utilizar la herencia para personalizar software existente. Cuando utilizamos la herencia para crear una nueva clase a partir de una clase existente, la nueva clase hereda los atributos y comportamientos de una clase existente, y después podemos agregar atributos y comportamientos o redefinir los comportamientos de una superclase para personalizar la clase con el objetivo de satisfacer nuestras necesidades.

Para los estudiantes puede resultar difícil apreciar los problemas que enfrentan los diseñadores y quienes implementan proyectos de software a gran escala para la industria. La gente experimentada en tales proyectos invariablemente afirma que una clave para mejorar el proceso de desarrollo de software es motivar la reutilización de software. La programación orientada a objetos en general, y Java en particular, ciertamente lo hacen.

Es la disponibilidad de bibliotecas substanciales y útiles la que proporciona los máximos beneficios de la reutilización de software a través de la herencia. Conforme se incremente el interés en Java, el interés en las bibliotecas de clases de Java se incrementará. Tal como el software producido por fabricantes independientes experimentó un gran crecimiento en la industria con la llegada de la computadora personal, así también sucederá con la creación y venta de las bibliotecas de clases de Java. Los diseñadores de aplicaciones construirán sus aplicaciones con estas bibliotecas, y los diseñadores de bibliotecas se verán recompensados al tener incluidas sus bibliotecas en las aplicaciones. Lo que vemos venir es un compromiso masivo a nivel mundial para el desarrollo de bibliotecas de clases de Java, para una amplia variedad de aplicaciones.

Observación de ingeniería de software 27.8



Crear una subclase no afecta el código fuente de su superclase, o el código en bytes de las superclases de Java; la integridad de una superclase se preserva a través de la herencia.

Una superclase especifica similitudes. Todas las clases derivadas de una superclase heredan las capacidades de esa superclase. En el proceso de diseño orientado a objetos, el diseñador busca similitudes entre un conjunto de clases y factores que necesita para formar superclases útiles. Las subclases entonces se personalizan más allá de las capacidades heredadas de las superclases.

Observación de ingeniería de software 27.9



Así como el diseñador de sistemas no orientados a objetos deben evitar la proliferación de funciones innecesarias, el diseñador de sistemas orientados a objetos debe evitar la proliferación de clases innecesarias. La proliferación de clases genera problemas de administración y puede dificultar la reutilización de software, simplemente porque es más difícil para un usuario potencial de una clase localizar esa clase en una amplia colección. El equilibrio se encuentra en crear pocas clases que proporcionen funcionalidad adicional importante, sin embargo, dichas clases pueden ser demasiado ricas para ciertos usuarios.

Tip de rendimiento 27.1



Si las clases producidas a través de la herencia son más grandes de lo necesario, podrían desperdiciarse recursos de memoria y de procesamiento. Herede de la clase “que más se acerque” a lo que usted necesita.

Observe que leer un conjunto de declaraciones de una subclase puede resultar confuso, ya que los miembros heredados no aparecen, pero dichos miembros están presentes en las subclases. Puede existir un problema similar en la documentación de las subclases.

Observación de ingeniería de software 27.10



En un sistema orientado a objetos, con frecuencia las clases se encuentran muy relacionadas. “Ubique” los atributos y comportamientos comunes y colóquelos en una superclase. Después utilice la herencia para formar subclases para que no tenga que repetir atributos y comportamientos comunes.

Observación de ingeniería de software 27.11



Las modificaciones a una superclase no requieren que las subclases se modifiquen, mientras la interfaz pública de la superclase permanezca sin cambios.

27.7 Composición versus herencia

Hemos explicado las relaciones *es un* que se implementan por herencia. También hemos explicado las relaciones *tiene un* (en ejemplos de capítulos anteriores) en la que una clase puede tener como miembros objetos de

otras clases; tales relaciones crean nuevas clases por medio de la *composición* de clases existentes. Por ejemplo, dadas las clases **Empleado**, **FechaNacimiento** y **NumeroTelefonico**, es inadecuado decir que un **Empleado** es una **FechaNacimiento** o que un **Empleado** es un **NumeroTelefonico**. Sin embargo, ciertamente es adecuado decir que un **Empleado** tiene una **FechaNacimiento** y que tiene un **NumeroTelefonico**.

27.8 Introducción al polimorfismo

Con el *polimorfismo*, es posible diseñar e implementar sistemas que sean más fácilmente *extensibles*. Los programas pueden escribirse para procesar genéricamente (como objetos de superclases) objetos de todas las clases existentes en una jerarquía. Las clases que no existen durante el desarrollo de un programa pueden agregarse con pocas o ninguna modificación a la parte genérica del programa; mientras esas clases sean parte de la jerarquía que se está procesando genéricamente. Las únicas partes de un programa que necesitan modificaciones son aquellas que requieran un conocimiento directo de una clase en particular que se agrega a la jerarquía. Estudiaremos dos jerarquías de clases importantes, y mostraremos cómo se manipulan de manera polimórfica los objetos a través de esas jerarquías.

27.9 Campos de tipo e instrucciones switch

Una manera de lidiar con objetos de diferentes tipos es por medio de una instrucción **switch** que realice la acción adecuada sobre cada objeto, basándose en el tipo de cada objeto. Por ejemplo, en una jerarquía de figuras en las que cada una tiene una variable de instancia **tipoFigura**, una estructura **switch** podría determinar a cuál método **print** llamar, basándose en el **tipoFigura** del objeto.

Existen muchos problemas con el uso de la lógica de **switch**. El programador podría olvidar hacer una prueba de tipos, cuando uno está garantizado. El programador podría olvidar probar todos los casos posibles de un **switch**. Si se modifica un sistema basado en **switch** agregando nuevos tipos, el programador podría olvidar insertar los nuevos casos en instrucciones **switch** existentes. Toda adición o eliminación de una clase demanda que cada instrucción **switch** en el sistema se modifique; rastreárlas puede llevarse demasiado tiempo y es propenso a errores.

Como veremos, la programación polimórfica puede eliminar la necesidad de la lógica de **switch**. El programador puede utilizar el mecanismo del polimorfismo de Java para realizar la lógica equivalente, con lo que eliminaría los tipos de errores generalmente asociados con la lógica de **switch**.

Tip para prevenir errores 27.2



Una consecuencia interesante de utilizar el polimorfismo es que los programas adquieran una apariencia simplificada; contienen menos lógica de separación, a favor de un código secuencial más sencillo. Esta simplificación facilita el probar, depurar y mantener un programa.

27.10 Método de vinculación dinámica

Suponga que un conjunto de clases de figuras como **Circulo**, **Triangulo**, **Rectangulo**, **Cuadrado**, etcétera, se derivan de la superclase **Figura**. En la programación orientada a objetos, cada una de estas clases puede dotarse con la habilidad de dibujarse a sí mismas. Cada clase tiene su propio método **draw**, y la implementación del método **draw** para cada figura es muy diferente. Cuando se dibuja una figura, cualquiera que ésta sea, sería bueno poder tratar a todas las figuras de manera genérica, como objetos de la superclase **Figura**. Después, para dibujar cualquier figura, podríamos simplemente llamar al método **draw** de la superclase **Figura**, y dejar al programa que determine dinámicamente (es decir, en tiempo de ejecución) cuál método **draw** de subclase utilizar, basándose en el tipo real del objeto.

Para permitir este tipo de comportamiento, declaramos **draw** en la superclase, y después redefinimos **draw** en cada una de las subclases para dibujar la figura adecuada.

Observación de ingeniería de software 27.12



Cuando una subclase elige no redefinir un método, la subclase simplemente hereda la definición del método de su superclase inmediata.

Si utilizamos una referencia de superclase para hacer referencia a un objeto de subclase e invocamos el método **draw**, el programa elegirá de manera dinámica (es decir, en tiempo de ejecución) el método **draw** de la subclase correcta. A esto se le llama *método de vinculación dinámica*, y lo ejemplificaremos en los ejemplos prácticos de este capítulo.

27.11 Métodos y clases final

Las variables pueden declararse como **final** para indicar que no pueden modificarse después de que se declaran, y que deben inicializarse cuando se declaran. También es posible definir métodos y clases con el modificador **final**.

Un método que se declara **final** no puede redefinirse en una subclase. Los métodos que se declaran como **static** y los métodos que se declaran como **private**, son implícitamente **final**. La definición de un método **final** nunca puede cambiar, por lo que el compilador puede optimizar el programa eliminando las llamadas a métodos **final**, y reemplazarlas con el código ampliado con sus definiciones en cada ubicación de las llamadas al método; una técnica conocida como *poner en línea al código*.

Una clase que se declara como **final** no puede ser una superclase (es decir, una clase no puede heredar de una clase **final**). Todos los métodos de una clase **final** son implícitamente **final**.

Tip de rendimiento 27.2



El compilador puede decidir poner en línea a una llamada a un método **final**, y lo hará para métodos **final** pequeños y sencillos. Colocarlas en línea no viola el encapsulamiento o el ocultamiento de información (pero mejora el rendimiento, ya que elimina la sobrecarga de realizar una llamada a un método).

Tip de rendimiento 27.3



Los preprocesadores canalizados pueden mejorar el rendimiento ejecutando simultáneamente diversas partes de las siguientes instrucciones, pero no si esas instrucciones siguen a una llamada a un método. Colocar en línea al código (lo que el compilador realiza en un método **final**) puede mejorar el rendimiento de estos preprocesadores, ya que elimina la transferencia de control fuera de línea asociada con una llamada a un método.

Observación de ingeniería de software 27.13



Una clase definida como **final** no puede extenderse, y cada uno de sus métodos es implícitamente **final**.

27.12 Superclases abstractas y clases concretas

Cuando pensamos en una clase como un tipo, asumimos que los objetos de ese tipo serán instanciados. Sin embargo, existen casos en los que resulta útil definir clases cuyos objetos nunca intentará instanciar el programador. Dichas clases se conocen como *clases abstractas* y contienen uno o más métodos abstractos. Éstas se utilizan como superclases en situaciones de herencia, por lo que normalmente nos referimos a ellas como *superclases abstractas*. Ningún objeto de superclases abstractas pueden instanciarse.

Error común de programación 27.7



Intentar crear una instancia de un objeto de una clase abstracta (es decir, una clase que contiene uno o más métodos abstractos), es un error de sintaxis.

Observación de ingeniería de software 27.14



Una clase abstracta puede tener datos de instancia y métodos no abstractos sujetos a las reglas normales de la herencia de las subclases. Una clase abstracta también pueden tener constructores.

El único propósito de una clase abstracta es proporcionar una superclase apropiada de la que otras clases puedan heredar la interfaz y/o la implementación (en un momento veremos ejemplos de esto). Las clases cuyos objetos pueden instanciarse se conocen como *clases concretas*.

Observación de ingeniería de software 27.15



Si una subclase se deriva de una superclase con un método **abstract**, y si no se proporciona una definición en la subclase para ese método **abstract** (es decir, si no se redefine ese método en la subclase), ese método permanece como **abstract** en la subclase. Como consecuencia, la subclase también es una clase **abstract**, y debe declararse explícitamente como **abstract**.

Observación de ingeniería de software 27.16



La habilidad de declarar un método `abstract` le da al diseñador de la clase suficiente poder sobre cómo implementará las subclases en una jerarquía de clases. Cualquier clase nueva que quiera heredar de esta clase es forzada a redefinir el método `abstract` (ya sea directamente o heredando de una clase que ha redefinido el método). De lo contrario, esa nueva clase contendrá un método `abstract` y, por lo tanto, será una clase `abstract` incapaz de instanciar objetos.

Podríamos tener una superclase abstracta **ObjetoBidimensional** y derivar clases concretas como **Cuadrado**, **Círculo**, **Triangulo**, etcétera. También podríamos tener una superclase abstracta **ObjetoTridimensional** y derivar clases concretas como **Cubo**, **Esfera**, **Cilindro**, etcétera. Las superclases abstractas son demasiado genéricas para definir objetos reales; necesitamos ser más específicos antes de que podamos pensar en instanciar objetos. Por ejemplo, si alguien le pide que “dibuja la figura”, ¿cuál dibujaría? Las clases concretas proporcionan las especificaciones que hacen razonable el crear instancias de objetos.

Se hace que una clase sea abstracta declarándola con la palabra reservada **abstract**. Una jerarquía no necesita contener ninguna clase **abstract**, pero como veremos, muchos buenos sistemas orientados a objetos tienen jerarquías de clases encabezadas por superclases **abstract**. En algunos casos, las clases abstractas constituyen la cima de algunos niveles de la jerarquía. Un buen ejemplo de esto es la jerarquía de figuras de la figura 27.2. La jerarquía comienza con la superclase **abstract Figura**. En el siguiente nivel hacia abajo tenemos otras dos superclases abstractas, a saber, **FiguraBidimensional** y **FiguraTridimensional**. El siguiente nivel hacia abajo comenzaría definiendo las clases concretas para las figuras bidimensionales como **Círculo** y **Cuadrado**, y clases concretas para las figuras tridimensionales como **Esfera** y **Cubo**.

Error común de programación 27.8



Si una clase con uno o más métodos `abstract` no se declara específicamente como `abstract`, es un error de sintaxis.

27.13 Ejemplo de polimorfismo

Aquí le presentamos un ejemplo de polimorfismo. Si una clase **Rectángulo** se deriva de la clase **Cuadrilatero**, entonces un objeto **Rectángulo** es una versión más específica del objeto **Cuadrilatero**. Una operación (como el cálculo del perímetro o el área) que puede realizarse sobre un objeto de la clase **Cuadrilatero** también puede realizarse sobre un objeto de la clase **Rectángulo**. Tales operaciones también pueden realizarse sobre otros “tipos de” **Cuadrilateros**, como **Cuadrados**, **Paralelogramos** y **Trapezoides**. Cuando se hace una solicitud para utilizar un método a través de una referencia de superclase, Java elige el método correcto redefinido de manera polimórfica en la subclase adecuada asociada con el objeto.

A través del polimorfismo, una llamada a un método puede provocar diferentes acciones, de acuerdo con el tipo del objeto que recibe la llamada. Esto da al programador una tremenda capacidad de expresión. En las siguientes secciones veremos el poder del polimorfismo.

Observación de ingeniería de software 27.17



Con el polimorfismo, el programador puede lidiar con las generalidades y deja que el ambiente en tiempo de ejecución se ocupe de lo específico. El programador puede ordenar que una amplia variedad de objetos se comporten de manera apropiada sin siquiera conocer los tipos de esos objetos.

Observación de ingeniería de software 27.18



El polimorfismo promueve la extensibilidad: El software escrito para invocar un comportamiento polimórfico se escribe de manera independiente a los tipos de los objetos a los que se envían los mensajes (es decir, llamadas a métodos). Por lo tanto, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en tales sistemas sin modificar el sistema base.

Observación de ingeniería de software 27.19



Si un método se declara como `final`, éste no puede redefinirse en las subclases, por lo que las llamadas al método no pueden enviarse de manera polimórfica a los objetos de esas subclases. La llamada al método aún puede enviarse a las subclases, pero responderán de manera idéntica, en lugar de hacerlo de manera polimórfica.



Observación de ingeniería de software 27.20

Una clase **abstract** define una interfaz común para los diversos miembros de una jerarquía de clase. La clase **abstract** contiene métodos que se definirán en las subclases. Todas las clases de la jerarquía pueden utilizar esta misma interfaz a través del polimorfismo.

Aunque no podemos crear instancias de objetos de superclases **abstract**, podemos declarar referencias a superclases **abstract**. Tales referencias pueden utilizarse para permitir manipulaciones polimórficas de objetos de subclases cuando tales objetos se instancian a partir de clases concretas.

Ahora consideremos más aplicaciones del polimorfismo. Un administrador de pantalla necesita desplegar una variedad de objetos, incluso nuevos tipos de objetos que se agregarán al sistema después de que esté escrito el administrador de pantalla. El sistema puede necesitar desplegar varias figuras (es decir, la superclase es **Figura**) como **Circulo**, **Triangulo**, **Rectangulo**, etcétera (cada clase de figura se deriva de la superclase **Figura**). El administrador de pantalla utiliza referencias de superclase (hacia **Figura**) para manipular los objetos a desplegar. Para dibujar cualquier objeto (independientemente del nivel en el que ese objeto apareza en la jerarquía de herencia), el administrador de pantalla utiliza una referencia de superclase hacia el objeto, y simplemente envía un mensaje **dibujar** al objeto. El método **dibujar** se declaró como **abstract** en la superclase **Figura** y se redefinió en cada una de las subclases. Cada objeto de **Figura** sabe cómo dibujarse a sí mismo. El administrador de pantalla no tiene que preocuparse por el tipo de cada objeto, o si el administrador de pantalla ha visto antes objetos de ese tipo; el administrador simplemente le indica a cada objeto que se dibuje.

El polimorfismo es particularmente efectivo para implementar sistemas de software en capas. Por ejemplo, en sistemas operativos, cada tipo de dispositivo físico puede funcionar de manera muy diferente. Incluso, los comandos *leer* o *escribir* datos desde y hacia los dispositivos pueden tener cierta uniformidad. El mensaje *escribir* enviado a un objeto controlado por un dispositivo necesita interpretarse específicamente en el contexto de ese controlador de dispositivo, y en cómo es que ese controlador manipula los dispositivos de un tipo específico. Sin embargo, la llamada a *escribir* misma, en realidad no es diferente de *escribir* en cualquier otro dispositivo del sistema; simplemente coloca cierto número de bytes de la memoria en ese dispositivo. Un sistema operativo orientado a objetos podría utilizar una superclase **abstract** para proporcionar una interfaz adecuada para todos los controladores de dispositivos. Entonces, a través de la herencia de esa superclase **abstract**, se forman las subclases para que funcionen de manera similar. Las capacidades (es decir, la interfaz **public**) ofrecidas por los controladores de dispositivos se proporcionan como métodos **abstract** en la superclase **abstract**. Las implementaciones de estos métodos **abstract** se proporcionan en las subclases que corresponden a los tipos específicos de los controladores de dispositivos.

En la programación orientada a objetos es común definir una *clase iteradora* que pueda recorrer todos los objetos de un contenedor (como un arreglo). Por ejemplo, si desea imprimir una lista de objetos de una lista ligada, puede crearse una instancia de un objeto iterador que devuelva el siguiente elemento de la lista ligada cada vez que se llame al iterador. Los iteradores comúnmente se utilizan en la programación polimórfica para recorrer un arreglo o una lista ligada de objetos desde varios niveles de una jerarquía. Las referencias de dicha lista serían referencias de superclase. Una lista de objetos de una superclase de la clase **FiguraBidimensional** podría contener objetos de las clases **Cuadrado**, **Circulo**, **Triangulo**, etcétera. Enviar un mensaje **dibujar** a cada objeto de la lista podría, por medio del polimorfismo, dibujar la imagen correcta en la pantalla.

27.14 Nuevas clases y vinculación dinámica

El polimorfismo ciertamente funciona bien cuando todas las clases posibles se conocen por adelantado. Sin embargo, también funciona cuando se agregan nuevos tipos de clases a los sistemas.

Las nuevas clases se acomodan por medio del método de la vinculación dinámica (también llamada *vinculación tardía*). No es necesario conocer el tipo de un objeto en tiempo de compilación para que una llamada polimórfica se compile. En tiempo de ejecución, la llamada se hace coincidir con el método del objeto llamado.

Un programa de administración de pantalla ahora puede manejar (sin tener que recomilar) nuevos tipos para desplegar objetos, conforme se agregan al sistema. La llamada al método **dibujar** permanece igual. Los nuevos objetos por sí mismos contienen un método **dibujar** que implementa las capacidades reales de dibujo.

Esto facilita el agregar nuevas capacidades al sistema con un impacto mínimo. También promueve la reutilización de software.

Tip de rendimiento 27.4



Cuando el polimorfismo se implementa con el método de vinculación dinámica, es eficiente.

Tip de rendimiento 27.5



Los tipos de manipulaciones polimórficas que se hacen posibles con la vinculación dinámica, también pueden lograrse por medio de la lógica de **switch** codificada manualmente, de acuerdo con los campos de tipo de los objetos. El código polimórfico generado por el compilador de Java se ejecuta con un rendimiento comparable con la lógica de **switch** eficientemente codificada.

27.15 Ejemplo práctico: Herencia de interfaz y de implementación

Ahora consideremos un ejemplo importante de herencia. Consideraremos la jerarquía **Punto**, **Círculo**, **Cilindro** de la figura 27.4. Como cabeza de la jerarquía tenemos a la superclase **abstract Figura**. Esta jerarquía mecánicamente demuestra el poder del polimorfismo. En los ejercicios, exploramos una jerarquía de figuras más realista.

```

1 // Figura 27.4: Figura.java
2 // Definición de la clase base abstracta Figura
3
4 public abstract class Figura extends Object {
5     public double area() { return 0.0; }
6     public double volumen() { return 0.0; }
7     public abstract String obtieneNombre();
8 } // fin de la clase Figura

```

Figura 27.4 Jerarquía Figura, Punto, Círculo, Cilindro; **Figura.java**.

```

9 // Figura 27.4: Punto.java
10 // Definición de la clase Punto
11
12 public class Punto extends Figura {
13     protected int x, y; // coordenadas del Punto
14
15     // constructor sin argumentos
16     public Punto() { establecePunto( 0, 0 ); }
17
18     // constructor sin argumentos
19     public Punto( int a, int b ) { establecePunto( a, b ); }
20
21     // Establece las coordenada x y y de Punto
22     public void establecePunto( int a, int b )
23     {
24         x = a;
25         y = b;
26     } // fin del método establecePunto
27
28     // obtiene la coordenada x
29     public int obtieneX() { return x; }
30
31     // obtiene la coordenada x
32     public int obtieneY() { return y; }
33

```

Figura 27.4 Jerarquía Figura, Punto, Círculo, Cilindro; **Punto.java**. (Parte 1 de 2.)

```

34     // convierte el punto a una representación String
35     public String toString()
36     { return "[" + x + ", " + y + "]"; }
37
38     // devuelve el nombre de la clase
39     public String obtieneNombre() { return "Punto"; }
40 } // fin de la clase Punto

```

Figura 27.4 Jerarquía Figura, Punto, Circulo, Cilindro; **Punto.java**. (Parte 2 de 2.)

```

41 // Figura 27.4: Circulo.java
42 // Definición de la clase Circulo
43
44 public class Circulo extends Punto { // hereda de Punto
45     protected double radio;
46
47     // constructor sin argumentos
48     public Circulo()
49     {
50         // llamada implícita al constructor de la superclase
51         estableceRadio( 0 );
52     } // fin del constructor Circulo
53
54     // Constructor
55     public Circulo( double r, int a, int b )
56     {
57         super( a, b ); // llama al constructor de la superclase
58         estableceRadio( r );
59     } // fin del constructor Circulo
60
61     // Establece el radio del Circulo
62     public void estableceRadio( double r )
63     { radio = ( r >= 0 ? r : 0 ); }
64
65     // Obtiene el radio del Circulo
66     public double obtieneRadio() { return radio; }
67
68     // Calcula el área del Circulo
69     public double area() { return Math.PI * radio * radio; }
70
71     // convierte Circulo a una String
72     public String toString()
73     { return "Centro = " + super.toString() +
74         "; Radio = " + radio; }
75
76     // devuelve el nombre de la clase
77     public String obtieneNombre() { return "Circulo"; }
78 } // fin de la clase Circulo

```

Figura 27.4 Jerarquía Figura, Punto, Circulo, Cilindro; **Circulo.java**.

```

79 // Figura 27.4: Cilindro.java
80 // Definición de la clase Cilindro
81
82 public class Cilindro extends Circulo {

```

Figura 27.4 Jerarquía Figura, Punto, Circulo, Cilindro; **Cilindro.java**. (Parte 1 de 2.)

```

83     protected double altura; // altura del Cilindro
84
85     // constructor sin argumentos
86     public Cilindro()
87     {
88         // llamada implícita al constructor de la superclase
89         estableceAltura( 0 );
90     } // fin del constructor Cilindro
91
92     // constructor
93     public Cilindro( double h, double r, int a, int b )
94     {
95         super( r, a, b ); // llama al constructor de la superclase
96         estableceAltura( h );
97     } // fin del constructor Cilindro
98
99     // Establece la altura del Cilindro
100    public void estableceAltura( double h )
101    {
102        altura = ( h >= 0 ? h : 0 );
103    }
104
105    // Obtiene la altura del Cilindro
106    public double obtieneAltura() { return altura; }
107
108    // Calcula el área del Cilindro (es decir, la superficie)
109    public double area()
110    {
111        return 2 * super.area() +
112            2 * Math.PI * radio * altura;
113    } // fin del método area
114
115    // Calcula el volumen del Cilindro
116    public double volumen() { return super.area() * altura; }
117
118    // Convierte un Cilindro a una String
119    public String toString()
120    {
121        return super.toString() + "; Altura = " + altura;
122    }
123 } // fin de la clase Cilindro

```

Figura 27.4 Jerarquía Figura, Punto, Circulo, Cilindro; **Cilindro.java**. (Parte 2 de 2.)

```

123 // Figura 27.4: Prueba.java
124 // Controlador para la jerarquía Punto, Circulo, Cilindro
125 import javax.swing.JOptionPane;
126 import java.text.DecimalFormat;
127
128 public class Prueba {
129     public static void main( String args[] )
130     {
131         Punto punto = new Punto( 7, 11 );
132         Circulo circulo = new Circulo( 3.5, 22, 8 );
133         Cilindro cilindro = new Cilindro( 10, 3.3, 10, 10 );

```

Figura 27.4 Jerarquía Figura, Punto, Circulo, Cilindro; **Prueba.java**. (Parte 1 de 2.)

```

134
135     Figura arregloDeFiguras[];
136
137     arregloDeFiguras = new Figura[ 3 ];
138
139     // asigna arregloDeFiguras[0] al objeto de la subclase Punto
140     arregloDeFiguras[ 0 ] = punto;
141
142     // asigna arregloDeFiguras[1] al objeto de la subclase Circulo
143     arregloDeFiguras[ 1 ] = circulo;
144
145     // asigna arregloDeFiguras[2] al objeto de la subclase Cilindro
146     arregloDeFiguras[ 2 ] = cilindro;
147
148     String salida =
149         punto.obtieneNombre() + ":" + punto.toString() + "\n" +
150         circulo.obtieneNombre() + ":" + circulo.toString() + "\n" +
151         cilindro.obtieneNombre() + ":" + cilindro.toString();
152
153     DecimalFormat precision2 = new DecimalFormat( "0.00" );
154
155     // Realiza el ciclo a través de arregloDeFiguras e imprime el nombre,
156     // el área, y el volumen de cada objeto.
157     for ( int i = 0; i < arregloDeFiguras.length; i++ ) {
158         salida += "\n\n" +
159             arregloDeFiguras[ i ].obtieneNombre() + ":" + +
160             arregloDeFiguras[ i ].toString() +
161             "\nÁrea = " +
162             precision2.format( arregloDeFiguras[ i ].area() ) +
163             "\nVolumen = " +
164             precision2.format( arregloDeFiguras[ i ].volumen() );
165     } // end for
166
167     JOptionPane.showMessageDialog( null, salida,
168         "Demostración de polimorfismo",
169         JOptionPane.INFORMATION_MESSAGE );
170
171     System.exit( 0 );
172 } // fin de main
173 } // fin de la clase Prueba

```

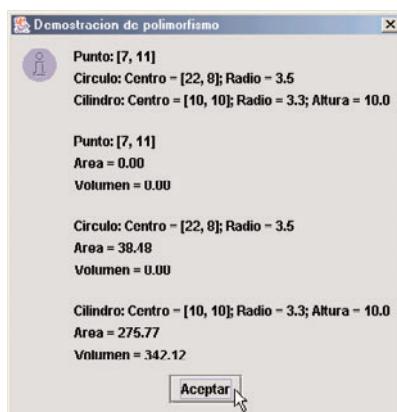


Figura 27.4 Jerarquía Figura, Punto, Circulo, Cilindro; **Prueba.java**. (Parte 2 de 2.)

Figura contiene el método `abstract obtieneNombre`, por lo que **Figura** debe declararse como una superclase `abstract`. **Figura** contiene otros dos métodos, `area` y `volumen`, cada uno de los cuales tiene una implementación que devuelve cero de manera predeterminada. **Punto** hereda estas implementaciones de **Figura**. Esto tiene sentido debido a que tanto el área como el volumen de un punto son cero. **Círculo** hereda el método `volumen` de **Punto**, pero **Círculo** proporciona su propia implementación del método `area`. **Cilindro** proporciona sus propias implementaciones de los métodos `area` (interpretada como la superficie del cilindro) y `volumen`.

En este ejemplo, la clase **Figura** se utiliza para definir un conjunto de métodos que todas las figuras de nuestra jerarquía tienen en común. Definir estos métodos en la clase **Figura** nos permite llamarlos de manera genérica a través de una referencia a **Figura**. Recuerde, los únicos métodos que pueden llamarse a través de cualquier referencia son los métodos públicos definidos en los tipos de clase declarados en la referencia y cualquier método público heredado en esa clase. Por lo tanto, podemos llamar a los métodos **Objeto** y **Figura**, a través de una referencia a **Figura**.

Observe que aunque **Figura** es una superclase `abstract`, aún contiene implementaciones de los métodos `area` y `volumen`, y estas implementaciones son heredables. La clase **Figura** proporciona una interfaz heredable (un conjunto de servicios) en la forma de tres métodos que todas las clases de la jerarquía contendrán. La clase **Figura** también proporciona algunas implementaciones que utilizarán las subclases de los primeros niveles de la jerarquía.

Este ejemplo práctico enfatiza que una subclase puede heredar la interfaz y/o la implementación de una superclase.

Observación de ingeniería de software 27.21



Las jerarquías diseñadas para la herencia de la implementación tienden a tener a su funcionalidad arriba en la jerarquía; cada nueva subclase hereda uno o más de los métodos que se definieron en una superclase, y utiliza las definiciones de la superclase.

Observación de ingeniería de software 27.22



Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad más abajo en la jerarquía; una superclase especifica uno o más métodos que deben invocarse de manera idéntica para cada objeto en la jerarquía (es decir, tienen la misma firma), pero las subclases individuales proporcionan sus propias implementaciones de los métodos.

La superclase **Figura** (figura 27.4, líneas 1 a 8) extiende a **Object**, consiste en tres métodos `public` y no contiene dato alguno (aunque podría). El método `obtieneNombre` es `abstract`, por lo que se redefine en las subclases. Los métodos `area` y `volumen` están definidos para que devuelvan `0.0`. Cuando es adecuado, estos métodos se redefinen en las subclases correspondientes a aquellas clases que tienen un cálculo de área diferente (clases **Círculo** y **Cilindro**) y/o un cálculo de volumen diferente (clase **Cilindro**).

La clase **Punto** (figura 27.4, líneas 9 a 40) se deriva de **Figura**. Un **Punto** tiene un área de `0.0` y un volumen de `0.0`, por lo que los métodos `area` y `volumen` de la superclase no se redefinen aquí; ellos se heredan como se definió en **Figura**. Otros métodos incluyen `establecePunto` para asignar nuevas coordenadas `x` y `y` a un **Punto**, y `obtieneX` y `obtieneY` para devolver las coordenadas `x` y `y` de un **Punto**. El método `obtieneNombre` es una implementación del método `abstract` en la superclase. Si no se definiera este método, la clase **Punto** sería una clase `abstract`.

La clase **Círculo** (figura 27.4, líneas 41 a 78) se deriva de **Punto**. Un **Círculo** tiene un volumen de `0.0`, por lo que el método de superclase `volumen` no se redefine; éste se hereda de la clase **Punto**, quien lo hereda de **Figura**. Un **Círculo** tiene un área diferente de un **Punto**, por lo que el método `area` se redefine. El método `obtieneNombre` es una implementación del método `abstract` de la superclase. Si este método no se redefine aquí, la versión de `obtieneNombre` de **Punto** se heredaría. Otros métodos incluyen `estableceRadio` para asignar un nuevo `radio` a un **Círculo**, y `obtieneRadio` para devolver el `radio` de un **Círculo**.

Observación de ingeniería de software 27.23



Una subclase siempre hereda la versión definida más recientemente de cada método `public` y `protected` de sus superclases directa e indirecta.

La clase **Cilindro** (figura 27.4, líneas 79 a 122) se deriva de **Circulo**. Un **Cilindro** tiene un área y un volumen diferente de aquellos de la clase **Circulo**, por lo que los métodos **area** y **volumen** se redefinen. El método **obtieneNombre** es una implementación del método **abstract** de la superclase. Si este método no se ha redefinido aquí, se hereda la versión de **obtieneNombre** de **Circulo**. Otros métodos incluyen **estableceAltura** para asignar una nueva **altura** a un **Cilindro**, y **obtieneAltura** para devolver la **altura** de un **Cilindro**.

El método **main** de la clase **Prueba** (figura 27.4, líneas 123 a 173) crea la instancia del objeto **punto** de **Punto**, del objeto **circulo** de **Circulo** y del objeto **cilindro** de **Cilindro** (líneas 131 a 133). Después, se instancia el arreglo **arregloDeFiguras** (línea 137). Este arreglo de referencias de la superclase **Figura** se referirá a cada objeto **instanciado** de la subclase. En la línea 140, la referencia **punto** se asigna al elemento **arregloDeFiguras[0]** del arreglo. En la línea 143, la referencia **circulo** se asigna al elemento **arregloDeFiguras[1]** del arreglo. En la línea 146, la referencia **cilindro** se asigna al elemento **arregloDeFiguras[2]** del arreglo. Ahora, cada referencia de la superclase **Figura** en el arreglo se refiere a un objeto de la subclase del tipo **Punto**, **Circulo** o **Cilindro**.

Las líneas 148 a 151 invocan a los métodos **obtieneNombre** y **toString** para ilustrar que los objetos se inicializan correctamente (vea las tres primeras líneas de la salida).

Después, la estructura **for** de las líneas 157 a 165 recorre el **arregloDeFiguras** y se hacen las siguientes llamadas, durante cada iteración del ciclo:

```
arregloDeFiguras[ i ].obtieneNombre()
arregloDeFiguras[ i ].toString()
arregloDeFiguras[ i ].area()
arregloDeFiguras[ i ].volumen()
```

Cada una de estas llamadas a métodos se invoca en el objeto al que **arregloDeFiguras** actualmente hace referencia. Cuando el compilador ve cada una de estas llamadas, simplemente intenta determinar si una referencia a **Figura** (**arregloDeFiguras[i]**) puede utilizarse para llamar a estos métodos. Para los métodos **obtieneNombre**, **area** y **volumen**, la respuesta es sí, ya que cada uno de estos métodos está definido en la clase **Figura**. Para el método **toString**, el compilador primero ve la clase **Figura** y determina que **toString** no está definido ahí, después el compilador continúa con la superclase **Figura (Object)** para determinar si **Figura** hereda un método **toString** que no tome argumentos (lo cual es cierto, ya que todos los **Objects** tienen un método **toString**).

La salida ilustra que los cuatro métodos se invocan adecuadamente, basándose en el tipo del objeto al que se hizo referencia. Primero, se despliega la cadena “**Punto:**” y las coordenadas del objeto **punto** (**arregloDeFiguras[0]**); el área y el volumen son 0. Después, se despliega la cadena “**Circulo:**”, las coordenadas del objeto **circulo**, y el **radio** del objeto **circulo** (**arregloDeFiguras[1]**); el área del **circulo** se calcula, y el volumen es 0. Por último, se despliega la cadena “**Cilindro:**”, las coordenadas del objeto **cilindro**, el **radio** del objeto **cilindro** y la **altura** del objeto **cilindro** (**arregloDeFiguras[2]**); el área y el volumen del **cilindro** se calculan. Todas las llamadas a los métodos **obtieneNombre**, **toString**, **area** y **volumen** se resuelven en tiempo de ejecución con vinculación dinámica.

27.16 Ejemplo práctico: Creación y uso de interfaces

Nuestro siguiente ejemplo (figura 27.5) reexamina la jerarquía **Punto**, **Circulo**, **Cilindro**, y reemplaza a la superclase **abstract** **Figura** con la interfaz **Figura**. Una definición de interfaz comienza con la palabra reservada **interface** y contiene un conjunto de métodos **public** y **abstract**. Las interfaces también pueden contener datos **public final static**. Para utilizar una interfaz, una clase debe especificar que la implementa y debe definir cada método en la interfaz con el número de argumentos y el tipo de retorno especificado en la definición de la interfaz. Si la clase deja indefinido un método en la interfaz, la clase se vuelve **abstract** y debe declararse como tal en la primera línea de la definición de su clase. Implementar una interfaz es como firmar un contrato con el compilador, el cual establece que “definiré todos los métodos especificados por la interfaz”.

Error común de programación 27.9



*Dejar indefinido un método de una interfaz, en una clase que implementa la interfaz, da como resultado un error de compilación que indica que la clase debe declararse como **abstract**.*

Por lo general, una interfaz se utiliza en lugar de una clase abstracta, cuando no hay una implementación predeterminada a heredar; es decir, no hay variables de instancia ni implementaciones predeterminadas de métodos. Como las clases **public abstract**, las interfaces en general son tipos de datos **public**, por lo que se definen a sí mismos en archivos con el mismo nombre que la interfaz y la extensión **.java**.

La definición de la interfaz **Figura** comienza en la línea 4. La interfaz **Figura** tiene los métodos **abstract area**, **volumen** y **obtieneNombre**. Como coincidencia, los tres métodos no toman argumentos. Sin embargo, éste no es un requerimiento de los métodos en una interfaz.

```

1 // Figura 27.5: Figura.java
2 // Definición de la interfaz Figura
3
4 public interface Figura {
5     public abstract double area();
6     public abstract double volumen();
7     public abstract String obtieneNombre();
8 } // fin de la interfaz Figura

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura**; **Figura.java**.

```

9 // Figura 27.5: Punto.java
10 // Definición de la clase Punto
11
12 public class Punto extends Object implements Figura {
13     protected int x, y; // coordenadas del Punto
14
15     // constructor sin argumentos
16     public Punto() { establecePunto( 0, 0 ); }
17
18     // constructor
19     public Punto( int a, int b ) { establecePunto( a, b ); }
20
21     // Establece las coordenadas x y y de Punto
22     public void establecePunto( int a, int b )
23     {
24         x = a;
25         y = b;
26     } // fin del método establecePunto
27
28     // obtiene la coordena x
29     public int obtieneX() { return x; }
30
31     // obtiene la coordena y
32     public int obtieneY() { return y; }
33
34     // convierte el punto a una representación a String
35     public String toString()
36     { return "[" + x + ", " + y + "]"; }
37
38     // devuelve el área
39     public double area() { return 0.0; }
40
41     // devuelve el volumen
42     public double volumen() { return 0.0; }

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura**; **Punto.java**. (Parte 1 de 2.)

```

43
44     // devuelve el nombre de la clase
45     public String obtieneNombre() { return "Punto"; }
46 } // fin de la clase Punto

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura; Punto.java**. (Parte 2 de 2.)

La línea 12

```
public class Punto extends Object implements Figura {
```

indica que la clase **Punto** extiende a la clase **Object** e implementa la interfaz **Figura**. La clase **Punto** proporciona definiciones de los tres métodos en la interfaz. El método **área** está definido en la línea 39, el método **volumen** está definido en la línea 42, y el método **obtieneNombre** está definido en la línea 45. Estos tres métodos satisfacen el requerimiento de la implementación para los tres métodos definidos en la interfaz. Hemos cumplido con el contrato con el compilador.

Cuando una clase implementa una interfaz, aplica la misma relación *es un* provista por la herencia. En nuestro ejemplo, la clase **Punto** implementa a **Figura**. Por lo tanto, un objeto **Punto** *es una Figura*. De hecho, los objetos de cualquier clase que extienden a **Punto**, también son objetos de **Figura**. A través de esta relación, hemos mantenido las definiciones originales de la clase **Circulo**, de la clase **Cilindro**, y de la clase de aplicación **Prueba** de la figura 27.4, para mostrar que se puede utilizar una interfaz en lugar de una clase **abstract**, para procesar de manera polimórfica unas **Figuras**. Observe que la salida del programa es idéntica a la de la figura 27.4. También observe que el método **toString** de **Object** es invocado a través de una referencia a la interfaz **Figura** (línea 166).

```

1 // Figura 27.5: Circulo.java
2 // Definición de la clase Circulo
3
4 public class Circulo extends Punto { // hereda desde Punto
5     protected double radio;
6
7     // constructor sin argumentos
8     public Circulo()
9     {
10         // llamada implícita al constructor de la superclase
11         estableceRadio( 0 );
12     } // fin del constructor Circulo
13
14     // Constructor
15     public Circulo( double r, int a, int b )
16     {
17         super( a, b ); // llamada al constructor de la superclase
18         estableceRadio( r );
19     } // fin del constructor Circulo
20
21     // Establece el radio del Circulo
22     public void estableceRadio( double r )
23     {
24         radio = ( r >= 0 ? r : 0 );
25     }
26     // Obtiene el radio del Circulo
27     public double obtieneRadio() { return radio; }
28
29     // Calcula el área del Círculo
30     public double area() { return Math.PI * radio * radio; }

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura; Circulo.java**. (Parte 1 de 2.)

```

30
31     // convierte el Circulo a una String
32     public String toString()
33         { return "Centro = " + super.toString() +
34             "; Radio = " + radio; }
35
36     // devuelve el nombre de la clase
37     public String obtieneNombre() { return "Circulo"; }
38 } // fin de la clase Circulo

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura; Circulo.java.**
(Parte 2 de 2.)

```

39 // Figura 27.5: Cilindro.java
40 // Definición de la clase Cilindro
41
42 public class Cilindro extends Circulo {
43     protected double altura; // altura del Cilindro
44
45     // constructor sin argumentos
46     public Cilindro()
47     {
48         // llamada implícita al constructor de la superclase
49         estableceAltura( 0 );
50     } // fin del constructor Cilindro
51
52     // constructor
53     public Cilindro( double h, double r, int a, int b )
54     {
55         super( r, a, b ); // llama al constructor de la superclase
56         estableceAltura( h );
57     } // fin del constructor Cilindro
58
59     // Establece la altura del Cilindro
60     public void estableceAltura( double h )
61     { altura = ( h >= 0 ? h : 0 ); }
62
63     // Obtiene la altura del Cilindro
64     public double obtieneAltura() { return altura; }
65
66     // Calcula el área del Cilindro (es decir, el área de la superficie)
67     public double area()
68     {
69         return 2 * super.area() +
70             2 * Math.PI * radio * altura;
71     } // fin del método area
72
73     // Calcula el volumen del Cilindro
74     public double volumen() { return super.area() * altura; }
75
76     // Convierte un Cilindro a una String
77     public String toString()
78     { return super.toString() + "; Altura = " + altura; }
79

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura; Cilindro.java.**
(Parte 1 de 2.)

```

80     // Devuelve el nombre de la clase
81     public String obtieneNombre() { return "Cilindro"; }
82 } // fin de la clase Cilindro

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz de **Figura; Cilindro.java**. (Parte 2 de 2.)



Observación de ingeniería de software 27.24

*Todos los métodos de la clase **Object** pueden invocarse por medio de una referencia a un tipo de dato interfaz; una referencia se refiere a un objeto, y todos los objetos tienen los métodos definidos por la clase **Object**.*

Un beneficio de utilizar interfaces es que una clase puede implementar tantas interfaces como sea necesario, además de extender una clase. Para implementar más de una interfaz, simplemente proporcione una lista separada por comas con los nombres de las interfaces, después de la palabra reservada **implements** en la definición de la clase. Esto es particularmente útil en el mecanismo de manipulación de eventos GUI. Una clase que implementa más de una interfaz que escucha eventos (como la **ActionListener** de los ejemplos anteriores) puede procesar diferentes tipos de eventos GUI, como veremos en el capítulo 29.

```

83 // Figura 27.5: Prueba.java
84 // Controlador para la jerarquía Punto, Circulo, Cilindro
85 import javax.swing.JOptionPane;
86 import java.text.DecimalFormat;
87
88 public class Prueba {
89     public static void main( String args[ ] )
90     {
91         Punto punto = new Punto( 7, 11 );
92         Circulo circulo = new Circulo( 3.5, 22, 8 );
93         Cilindro cilindro = new Cilindro( 10, 3.3, 10, 10 );
94
95         Figura arregloDeFiguras[];
96
97         arregloDeFiguras = new Figura[ 3 ];
98
99         // asigna arregloDeFiguras[0] al objeto de la subclase Punto
100        arregloDeFiguras[ 0 ] = punto;
101
102        // asigna arregloDeFiguras[0] al objeto de la subclase Circulo
103        arregloDeFiguras[ 1 ] = circulo;
104
105        // asigna arregloDeFiguras[0] al objeto de la subclase Cilindro
106        arregloDeFiguras[ 2 ] = cilindro;
107
108        String salida =
109            punto.obtieneNombre() + ": " + punto.toString() + "\n" +
110            circulo.obtieneNombre() + ": " + circulo.toString() + "\n" +
111            cilindro.obtieneNombre() + ": " + cilindro.toString();
112
113        DecimalFormat precision2 = new DecimalFormat( "#0.00" );
114
115        // Ciclo a través de arregloDeFiguras e impresión del nombre,
116        // el área, y el volumen de cada objeto.
117        for ( int i = 0; i < arregloDeFiguras.length; i++ ) {
118            salida += "\n\n" +

```

Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz **Figura; Prueba.java**. (Parte 1 de 2.)

```

119         arregloDeFiguras[ i ].obtieneNombre() + ": " +
120         arregloDeFiguras[ i ].toString() +
121         "\nArea = " +
122         precision2.format( arregloDeFiguras[ i ].area() ) +
123         "\nVolumen = " +
124         precision2.format( arregloDeFiguras[ i ].volumen() );
125     }
126
127     JOptionPane.showMessageDialog( null, salida,
128         "Demostracion de polimorfismo",
129         JOptionPane.INFORMATION_MESSAGE );
130
131     System.exit( 0 );
132 } // fin de main
133 } // fin de la clase Prueba

```

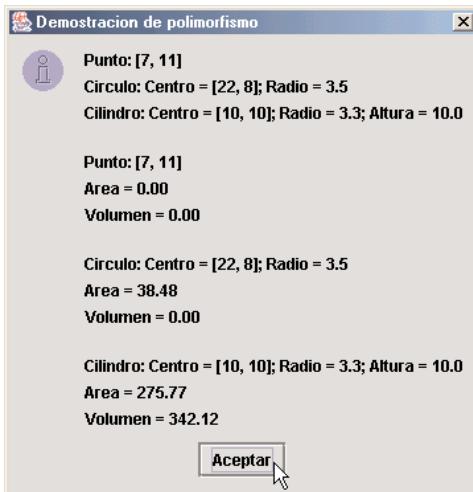


Figura 27.5 Jerarquía Punto, Circulo, Cilindro con una interfaz. **Figura: Prueba.java.** (Parte 2 de 2.)

Otro uso de las interfaces es definir un conjunto de constantes que pueden utilizarse en muchas definiciones de clases. Considere la interfaz **Constantes**

```

public interface Constantes {
    public static final int UNO = 1;
    public static final int DOS = 2;
    public static final int TRES = 3;
}

```

Las clases que implementan la interfaz **Constantes** pueden utilizar las constantes **UNO**, **DOS** y **TRES** en cualquier parte de la definición de la clase. Una clase puede incluso utilizar estas constantes, simplemente importando la interfaz, y después refiriéndose a cada constante como **Constantes.UNO**, **Constantes.DOS** y **Constantes.TRES**. Ningún método está declarado en esta interfaz, por lo que a una clase que implementa la interfaz no se le solicita que proporcione implementación alguna.

27.17 Definiciones de clases internas

Todas las definiciones de clases que hemos explicado hasta este punto, se definieron con alcance de archivo; las clases se definieron en archivos, pero no dentro de otras clases de esos archivos. Java proporciona una facilidad llamada *clases internas*, en las que las clases pueden definirse dentro de otras clases. Tales clases pueden ser definiciones completas de clases, o definiciones de *clases internas anónimas* (clases sin un nombre). Las clases

internas se utilizan principalmente en la manipulación de eventos. Sin embargo, tienen otros beneficios. Por ejemplo, cuando se define un tipo de dato abstracto cola, se puede utilizar una clase interna para representar los objetos que almacena cada elemento actualmente en la cola. Sólo la estructura de datos cola requiere saber cómo se almacenan los objetos de manera interna, por lo que la implementación puede ocultarse definiendo una clase interna como parte de la clase **Cola**.

Las clases internas con frecuencia se utilizan con la manipulación de eventos GUI, por lo que aprovechamos esta oportunidad, no sólo para mostrarle las definiciones de clases internas, sino para también demostrarle una aplicación que se ejecuta en su propia ventana. Una vez que complete este ejemplo, podrá utilizar en sus aplicaciones las técnicas GUI que hasta el momento hemos mostrado sólo en applets.

Para demostrar una definición de una clase interna, la figura 27.6 utiliza una versión simplificada de la clase **Hora2** (renombrada aquí como **Hora**) correspondiente a la figura 26.3. La clase **Hora** proporciona un constructor predeterminado, los mismos métodos *establecer/obtener* de la figura 26.3, y un método **toString**. Además, este programa define la clase **VentanaPruebaHora** como una aplicación. La aplicación se ejecuta en su propia ventana.

```
1 // Figura 27.6: Hora.java
2 // Definición de la clase Hora
3 import java.text.DecimalFormat; // se utiliza para dar formato a números
4
5 // Esta clase contiene la hora en formato de 24 horas
6 public class Hora extends Object {
7     private int hora; // 0 - 23
8     private int minuto; // 0 - 59
9     private int segundo; // 0 - 59
10
11    // el constructor Hora inicializa cada variable de instancia
12    // en cero. Asegura que el objeto Hora se encuentra en un
13    // estado consistente
14    public Hora() { estableceHora( 0, 0, 0 ); }
15
16    // Establece un nuevo valor de hora con el formato universal. Realiza
17    // las validaciones de los datos. Establece en cero los valores no válidos.
18    public void estableceHora( int h, int m, int s )
19    {
20        estableceHora( h ); // establece la hora
21        estableceMinuto( m ); // establece el minuto
22        estableceSegundo( s ); // establece el segundo
23    } // fin del método estableceHora
24
25    // establece la hora
26    public void estableceHora( int h )
27    {
28        hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
29
30        // establece el minuto
31        public void estableceMinuto( int m )
32        {
33            minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
34
35            // establece el segundo
36            public void estableceSegundo( int s )
37            {
38                segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
39
40                // obtiene la hora
41                public int obtieneHora() { return hora; }
42
43            } // fin del método estableceSegundo
44        } // fin del método estableceMinuto
45    } // fin del método estableceHora
46
47    // obtiene la hora
48    public int obtieneHora() { return hora; }
49
50} // fin de la clase Hora
```

Figura 27.6 Demostración de una clase interna en una aplicación de visualización; **Hora.java**. (Parte 1 de 2.)

```

39
40    // obtiene el minuto
41    public int obtieneMinuto() { return minuto; }
42
43    // obtiene el segundo
44    public int obtieneSegundo() { return segundo; }
45
46    // Conversión a una String en formato de hora estándar
47    public String toString()
48    {
49        DecimalFormat dosDigitos = new DecimalFormat( "00" );
50
51        return ( ( obtieneHora() == 12 || obtieneHora() == 0 ) ?
52                  12 : obtieneHora() % 12 ) + ":" +
53                  dosDigitos.format( obtieneMinuto() ) + ":" +
54                  dosDigitos.format( obtieneSegundo() ) +
55                  ( obtieneHora() < 12 ? " AM" : " PM" );
56    } // fin del método toString
57 } // fin de la clase Hora

```

Figura 27.6 Demostración de una clase interna en una aplicación de visualización; **Hora.java**.
(Parte 2 de 2.)

```

58 // Figura 27.6: VentanaPruebaHora.java
59 // Demostración de los métodos establecer y obtener de la clase Hora
60 import java.awt.*;
61 import java.awt.event.*;
62 import javax.swing.*;
63
64 public class VentanaPruebaHora extends JFrame {
65     private Hora h;
66     private JLabel etiquetaHora, etiquetaMinuto, etiquetaSegundo;
67     private JTextField campoHora, campoMinuto,
68                     campoSegundo, desplegar;
69     private JButton botonSalida;
70
71     public VentanaPruebaHora()
72     {
73         super( "Demostración de la clase Interna" );
74
75         h = new Hora();
76
77         Container c = getContentPane();
78
79         // crea una instancia de la clase interna
80         ActionEventHandler manipulador = new ActionEventHandler();
81
82         c.setLayout( new FlowLayout() );
83         etiquetaHora = new JLabel( "Establece la hora" );
84         campoHora = new JTextField( 10 );
85         campoHora.addActionListener( manipulador );
86         c.add( etiquetaHora );
87         c.add( campoHora );

```

Figura 27.6 Demostración de una clase interna en una aplicación de visualización;
VentanaPruebaHora.java.(Parte 1 de 3.)

```
88
89      etiquetaMinuto = new JLabel( "Establece el minuto" );
90      campoMinuto = new JTextField( 10 );
91      campoMinuto.addActionListener( manipulador );
92      c.add( etiquetaMinuto );
93      c.add( campoMinuto );
94
95      etiquetaSegundo = new JLabel( "Establece el segundo" );
96      campoSegundo = new JTextField( 10 );
97      campoSegundo.addActionListener( manipulador );
98      c.add( etiquetaSegundo );
99      c.add( campoSegundo );
100
101     desplegar = new JTextField( 30 );
102     desplegar.setEditable( false );
103     c.add( desplegar );
104
105     botonSalida = new JButton( "Salir" );
106     botonSalida.addActionListener( manipulador );
107     c.add( botonSalida );
108 } // fin del constructor VentanaPruebaHora
109
110 public void despliegaHora()
111 {
112     desplegar.setText( "La hora es: " + h );
113 } // fin del método despliegaHora
114
115 public static void main( String args[] )
116 {
117     VentanaPruebaHora ventana = new VentanaPruebaHora();
118
119     ventana.setSize( 400, 140 );
120     ventana.show();
121 } // fin de main
122
123 // Definición de la clase interna para la manipulación de eventos
124 private class ActionEventHandler implements ActionListener {
125     public void actionPerformed( ActionEvent e )
126     {
127         if ( e.getSource() == botonSalida )
128             System.exit( 0 ); // termina la aplicación
129         else if ( e.getSource() == campoHora ) {
130             h.estableceHora(
131                 Integer.parseInt( e.getActionCommand() ) );
132             campoHora.setText( "" );
133         }
134         else if ( e.getSource() == campoMinuto ) {
135             h.estableceMinuto(
136                 Integer.parseInt( e.getActionCommand() ) );
137             campoMinuto.setText( "" );
138         }
139         else if ( e.getSource() == campoSegundo ) {
140             h.estableceSegundo(
141                 Integer.parseInt( e.getActionCommand() ) );
```

Figura 27.6 Demostración de una clase interna en una aplicación de visualización;
VentanaPruebaHora.java. (Parte 2 de 3.)

```

142         campoSegundo.setText( " " );
143     }
144
145     despliegaHora();
146 } // fin del método accionRealizada
147 } // fin de la clase ManipDeEventos
148 } // fin de la clase VentanaPruebaHora

```

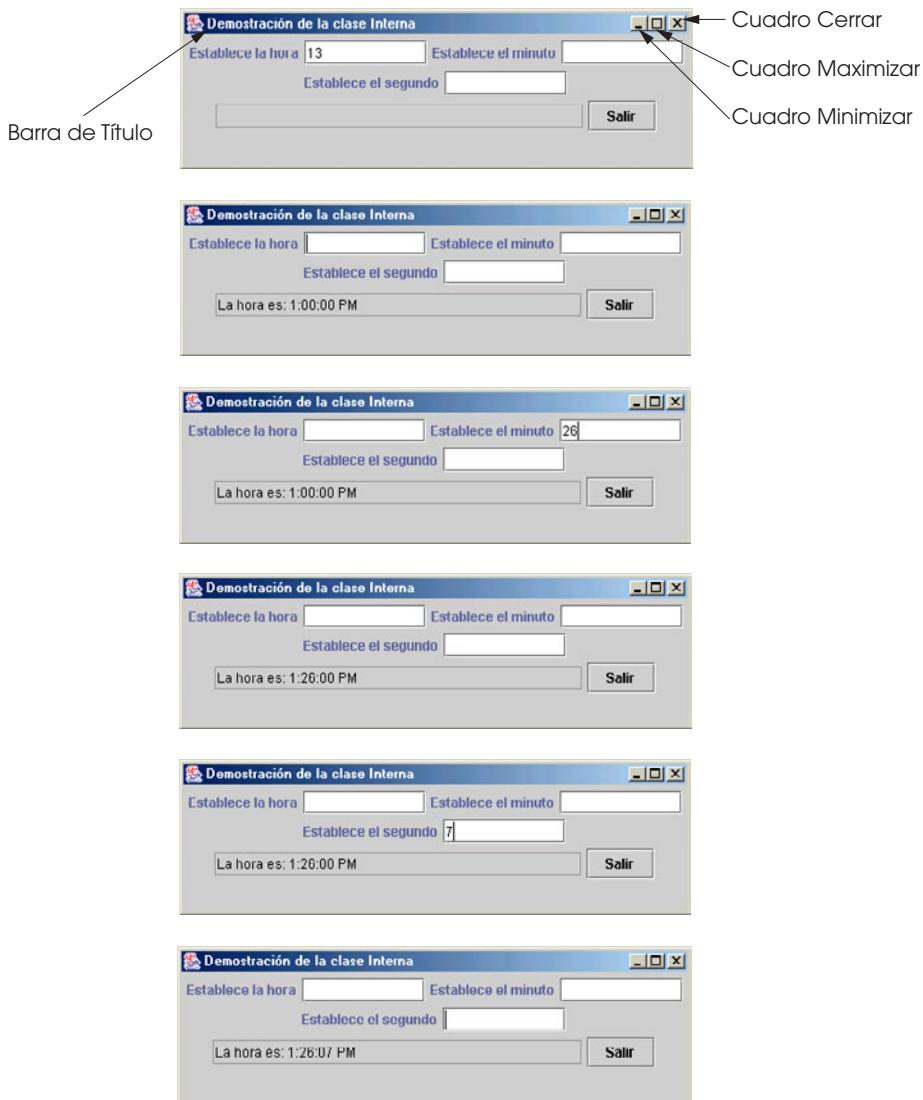


Figura 27.6 Demostración de una clase interna en una aplicación de visualización;
VentanaPruebaHora.java. (Parte 3 de 3.)

La línea 64

```
public class VentanaPruebaHora extends JFrame {
```

indica que la clase **VentanaPruebaHora** extiende a la clase **JFrame** (del paquete **javax.swing**), en lugar de la clase **JApplet** (como muestra la figura 26.3). La superclase **JFrame** proporciona los atributos y comportamientos básicos de una ventana; una *barra de título* y botones para *minimizar*, *maximizar* y *cerrar* la

ventana (todos etiquetados en la primera captura de pantalla). La clase **VentanaPruebaHora** utiliza los mismos componentes GUI que el applet de la figura 26.3, con la excepción de que al botón (línea 69) ahora se le llama **botonSalida**, y se utiliza para finalizar la aplicación.

El método **init** del applet se reemplazó con un constructor (línea 71) para garantizar que los componentes GUI de la ventana se crean conforme comienza la ejecución. El método **main** (línea 115) define un objeto **new** de la clase **VentanaPruebaHora** que da como resultado una llamada al constructor. Recuerde, **init** es un método especial, cuya invocación está garantizada cuando un applet comienza su ejecución. Sin embargo, este programa no es un applet, por lo que no se garantiza que el método **init** sea llamado.

En el constructor aparecen diversas características nuevas. La línea 73 llama al constructor de la superclase **JFrame** con la cadena “**Demostracion de una clase interna**”. Esta cadena se despliega en la barra de título de la ventana por medio del constructor de **JFrame**. La línea 80

```
ActionEventHandler manipulador = new ActionEventHandler();
```

define dos instancias de nuestra clase **ActionEventHandler** y la asigna a **manipulador**. Esta referencia se pasa a cada uno de las cuatro llamadas a **ActionListener** (líneas 85, 91, 97 y 106) que registran los manipuladores de eventos para cada componente GUI que genera los eventos en el ejemplo (**campoHora**, **campoMinuto**, **campoSegundo**, y **botonSalida**). Cada llamada a **addActionListener** requiere un objeto de **ActionListener** para pasarlo como argumento. En realidad, **manipulador** es un **ActionListener**. La línea 124 (la primera línea de la definición de la clase interna)

```
private class ActionEventHandler implements ActionListener {
```

indica que la clase interna **ActionEventHandler** implementa a **ActionListener**. Así, cada objeto de tipo **ActionEventHandler** es un **ActionListener**. ¡Se satisface el requerimiento que indica que **addActionListener** se pase como un objeto de tipo **ActionListener**! Ésta es una relación que se utiliza de manera extensiva en el mecanismo de manipulación de eventos del GUI, como lo verá en los siguientes capítulos. La clase interna se define como **private** debido a que solamente se utilizará en la definición de la clase. Las clases internas pueden ser **private**, **protected** o **public**.

Un objeto de la clase interna tiene una relación especial con el objeto de la clase externa que lo crea. Al objeto de la clase interna se le permite tener acceso directo a todas las variables de instancia y a los métodos del objeto de la clase externa. El método **actionPerformed** (línea 125) de la clase **EventHandler** hace justamente eso. A través del método, se utilizan las variables de instancia **h**, **botonSalida**, **campoHora**, **campoMinuto**, así como su método **despliegaHora**. Observe que ninguno de éstos es un “manipulador” del objeto de la clase externa. Ésta es una relación libre creada por el compilador entre la clase externa y sus clases internas.

Observación de ingeniería de software 27.25



A un objeto de la clase interna se le permite tener acceso directo a las variables de instancia y a los métodos del objeto de la clase externa que la define.

[Nota: Esta aplicación se debe finalizar al presionar el botón **Entrar**. Recuerde, una aplicación que despliega una ventana debe terminar con una llamada a **System.exit(0)**. Observe además que una ventana en Java tiene 0 pixeles de ancho y 0 pixeles de alto y no se despliega de manera predeterminada. Las líneas 119 y 120 redimensionan el tamaño de la ventana y la muestran en la pantalla.]

Una clase interna también puede definirse dentro del método de una clase. Tal clase interna tiene acceso a los miembros de la clase externa. Sin embargo, tiene acceso limitado a las variables locales del método en el cual está definida.

Observación de ingeniería de software 27.26



Una clase interna definida dentro de un método tiene acceso directo a todas las variables de instancia y métodos del objeto de la clase externa en la que se define y en cualquier variable local de **final** en el método.

La aplicación de la figura 27.7 modifica la clase **VentanaPruebaHora** para utilizar las *clases anónimas internas* definidas dentro de métodos. Una clase anónima interna no tiene nombre, de modo que se debe crear la clase anónima interna en el punto en donde se define la clase dentro del programa. En este ejemplo, demostramos las clases anónimas internas de dos formas. Primero, sepáramos las clases anónimas internas que

implementan una interfaz (**ActionListener**) para crear manipuladores para cada uno de los tres **JTextFields campoHora**, **campoMinuto** y **campoSegundo**. También demostramos cómo terminar una aplicación cuando el usuario hace clic en el cuadro **Cerrar** en la ventana. El manipulador de eventos se define como la clase anónima interna que extiende a la clase (**WindowAdapter**). La clase **Hora** es idéntica a la figura 27.6, de modo que no se incluye aquí. Además, el botón **Salir** se eliminó de este ejemplo.

```
1 // Figura 27.7: VentanaPruebaHora.java
2 // Demostración de los métodos establecer y obtener para la clase Hora
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class VentanaPruebaHora extends JFrame {
8     private Hora h;
9     private JLabel etiquetaHora, etiquetaMinuto, etiquetaSegundo;
10    private JTextField campoHora, campoMinuto,
11                    campoSegundo, despliega;
12
13    public VentanaPruebaHora()
14    {
15        super( "Demostración de la clase interna" );
16
17        h = new Hora();
18
19        Container c = getContentPane();
20
21        c.setLayout( new FlowLayout() );
22        etiquetaHora = new JLabel( "Establece la hora" );
23        campoHora = new JTextField( 10 );
24        campoHora.addActionListener(
25            new ActionListener() { // clase interna anónima
26                public void actionPerformed( ActionEvent e )
27                {
28                    h.estableceHora(
29                        Integer.parseInt( e.getActionCommand() ) );
30                    campoHora.setText( "" );
31                    despliega();
32                } // fin del método actionPerformed
33            } // fin de la clase interna anónima
34        ); // fin de addActionListener
35        c.add( etiquetaHora );
36        c.add( campoHora );
37
38        etiquetaMinuto = new JLabel( "Establece el minuto" );
39        campoMinuto = new JTextField( 10 );
40        campoMinuto.addActionListener(
41            new ActionListener() { // clase interna anónima
42                public void actionPerformed( ActionEvent e )
43                {
44                    h.estableceMinuto(
45                        Integer.parseInt( e.getActionCommand() ) );
46                    campoMinuto.setText( "" );
47                }
48            } // fin de la clase interna anónima
49        ); // fin de addActionListener
50        c.add( etiquetaMinuto );
51        c.add( campoMinuto );
52
53        etiquetaSegundo = new JLabel( "Establece el segundo" );
54        campoSegundo = new JTextField( 10 );
55        campoSegundo.addActionListener(
56            new ActionListener() { // clase interna anónima
57                public void actionPerformed( ActionEvent e )
58                {
59                    h.estableceSegundo(
60                        Integer.parseInt( e.getActionCommand() ) );
61                    campoSegundo.setText( "" );
62                }
63            } // fin de la clase interna anónima
64        ); // fin de addActionListener
65        c.add( etiquetaSegundo );
66        c.add( campoSegundo );
67
68        setContentPane( c );
69
70        pack();
71        setVisible( true );
72    }
73
74    public static void main( String[] args )
75    {
76        VentanaPruebaHora vp = new VentanaPruebaHora();
77    }
78}
```

Figura 27.7 Demostración de las clases anónimas internas; **VentanaPruebaHora.java**.
(Parte 1 de 3).

```
47             despliegaHora();
48         }
49     }
50 );
51 c.add( etiquetaMinuto );
52 c.add( campoMinuto );
53
54 etiquetaSegundo = new JLabel( "Establece el segundo" );
55 campoSegundo = new JTextField( 10 );
56 campoSegundo.addActionListener(
57     new ActionListener() { // clase interna anónima
58         public void actionPerformed( ActionEvent e )
59     {
60         h.estableceSegundo(
61             Integer.parseInt( e.getActionCommand() ) );
62         campoSegundo.setText( "" );
63         despliegaHora();
64     } // fin del método actionPerformed
65 } // fin de la clase interna anónima
66 ); // fin de addActionListener
67 c.add( etiquetaSegundo );
68 c.add( campoSegundo );
69
70 despliega = new JTextField( 30 );
71 despliega.setEditable( false );
72 c.add( despliega );
73 } // fin del constructor VentanaPruebaHora
74
75 public void despliegaHora()
76 {
77     despliega.setText( "La hora es: " + h );
78 } // fin del método despliegaHora
79
80 public static void main( String args[ ] )
81 {
82     VentanaPruebaHora ventana = new VentanaPruebaHora();
83
84     ventana.addWindowListener(
85         new WindowAdapter() {
86             public void windowClosing( WindowEvent e )
87             {
88                 System.exit( 0 );
89             } // fin del método windowClosing
90         } // fin de la clase interna anónima
91     ); // fin de addWindowListener
92
93     ventana.setSize( 400, 120 );
94     ventana.show();
95 } // fin de main
96 } // fin de la clase VentanaPruebaHora
```

Figura 27.7 Demostración de las clases anónimas internas; **VentanaPruebaHora.java**.
(Parte 2 de 3).

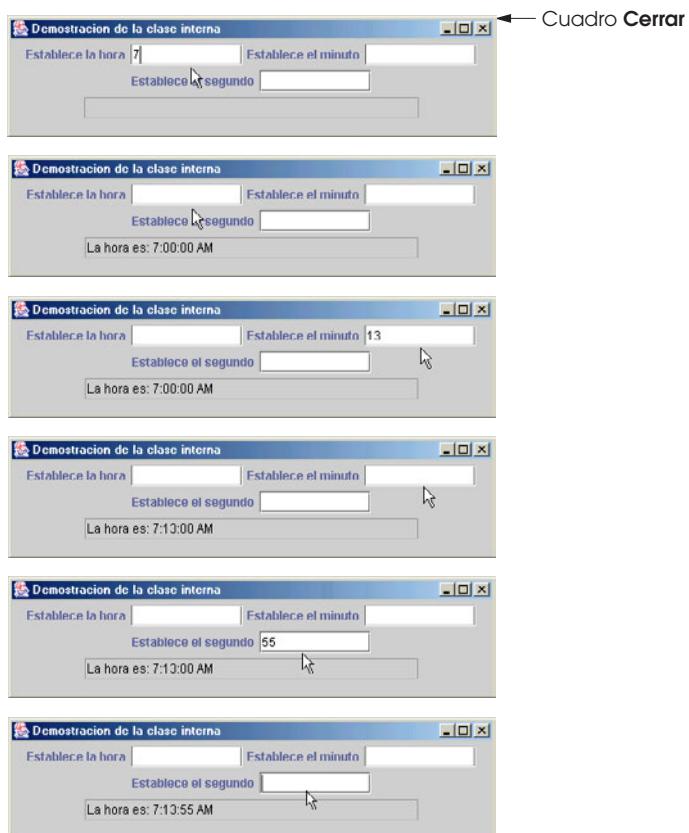


Figura 27.7 Demostración de las clases anónimas internas; **VentanaPruebaHora.java**.
(Parte 3 de 3).

Cada una de las tres **JTextField** que generan los eventos en este programa tiene una clase interna anónima para manipular los eventos, de modo que aquí solamente explicaremos la clase interna anónima para **campoHora**. Las líneas 24 a 34

```

campoHora.addActionListener(
    new ActionListener() { // clase interna anónima
        public void actionPerformed( ActionEvent e )
        {
            h.estableceHora(
                Integer.parseInt( e.getActionCommand() ) );
            campoHora.setText( " " );
            despliegaHora();
        } // fin del método actionPerformed
    } // fin de la clase interna anónima
); // fin de addActionListener

```

llama al método **campoHora** de **addActionListener**. El argumento para este método debe ser un objeto que *es un ActionListener* (es decir, cualquier objeto de la clase que implementa **ActionListener**). Las líneas 25 a 33 utilizan una sintaxis especial de Java para definir una clase anónima interna y crear un objeto de la clase que se pasa como el argumento de **ActionListener**. La línea 25

```
new ActionListener() { // clase interna anónima
```

utiliza el operador **new** para crear un objeto. La sintaxis **ActionListener()** inicia la definición de una clase interna anónima que implementa la interfaz **ActionListener**. Esto es similar a iniciar la definición de la clase como

```
public class MiManipulador implements ActionListener {
```

Los paréntesis después de **ActionListener** indican una llamada al constructor predeterminado de la clase anónima interna.

La llave izquierda de apertura ({}) al final de la línea 25 y la llave derecha de cierre (}) en la línea 33 definen el cuerpo de la clase. Las líneas 26 a 32 definen el método, **actionPerformed**, que se requiere en cualquier clase que implementa **ActionListener**. Se llama al método **ActionPerformed** cuando el usuario presiona *Entrar* mientras escribe en **campoHora**.

Observación de ingeniería de software 27.27



Cuando una clase anónima interna implementa una interfaz, la clase debe definir cada método en la interfaz.

El método **main** crea una instancia de la clase **VentanaPruebaHora** (línea 82), redimensiona la ventana (línea 93) y despliega la ventana (línea 94).

Windows genera distintos eventos que explicaremos en el capítulo 29. Para este ejemplo explicamos el evento generado cuando el usuario hace clic en el cuadro cerrar de la ventana, un *evento para cerrar la ventana*. Las líneas 84 a 91

```
ventana.addWindowListener(
    new WindowAdapter() {
        public void windowClosing( WindowEvent e )
        {
            System.exit( 0 );
        } // fin del método windowClosing
    } // fin de la clase interna anónima
); // fin de addWindowListener
```

permite al usuario terminar la aplicación al hacer clic en el cuadro **Cerrar** de la ventana (etiquetado en la primera pantalla de captura). El método **addWindowListener** registra un receptor de eventos de la ventana. El argumento **addWindowListener** debe ser una referencia a un objeto que *es un WindowListener* (paquete **java.awt.event**) (es decir, cualquier objeto de la clase que implementa **WindowListener**). Sin embargo, existen siete métodos diferentes que se deben definir en cada clase que implementa **WindowListener** y solamente necesitamos uno en este ejemplo, **WindowClosing**. Para las interfaces con más de un método, Java proporciona una clase correspondiente (llamada *clase adaptadora*) que de antemano implementa todos los métodos en la interfaz para usted. Todo lo que necesita hacer es extender la clase adaptadora y redefinir los métodos requeridos en su programa.

Error común de programación 27.10



Extender una clase adaptadora y escribir incorrectamente el nombre de un método que va a redefinir, es un error de lógica.

Las líneas 85 a 90 utilizan una sintaxis especial de Java para definir una clase interna anónima y crear un objeto de la clase que se pasa como el argumento de **addWindowListener**. La línea 85

```
new WIndowAdapter() {
```

utiliza el operador **new** para crear un objeto. La sintaxis de **WindowAdapter()** comienza la definición de la clase que extiende a la clase **WindowAdapter**. Esto es similar al inicio de la definición de la clase

```
public class MiManipulador extiende WindowAdapter {
```

El paréntesis después de **WindowAdapter** indica una llamada al constructor predeterminado de la clase anónima interna. La clase **WindowAdapter** implementa la interfaz **WindowListener**, el tipo exacto requerido para el argumento de **addWindowListener**.

La llave izquierda de cierre ({{}) al final de la línea 85 y la llave derecha de cierre (}) en la línea 90 definen el cuerpo de la clase. Las líneas 86 a 89 redefinen el método de **WindowAdapter**, **windowClosing**, que se llama cuando el usuario hace clic en el cuadro **Cerrar** de la ventana. En este ejemplo, **windowClosing** termina la aplicación con una llamada a **System.exit(0)**.

En los dos últimos ejemplos, vimos que las clases internas se pueden utilizar para crear manipuladores de eventos, y que las clases internas anónimas pueden definirse para manejar eventos de manera individual para cada componente GUI. En el capítulo 29, volveremos a revisar este concepto conforme expliquemos con detalle el mecanismo de manipulación de eventos.

27.18 Notas sobre las definiciones de clases internas

Esta sección presenta diversas notas de interés para los programadores con respecto a la definición y el uso de clases internas.

1. Compilar una clase que contiene clases internas da como resultado archivos separados `.class` para cada clase. Las clases internas con nombres tienen el nombre de archivo `NombreClaseExterna$NombreClaseInterna.class`. Las clases internas anónimas tienen el nombre de archivo `NombreClaseExterna$#.class`, donde # comienza en 1 y se incrementa para cada clase anónima que se encuentre durante la compilación.
2. Las clases internas con nombres de clases pueden definirse como `public`, `protected`, de acceso a paquetes o `private`, y están sujetas a las mismas restricciones de uso que los otros miembros de una clase.
3. Para acceder a la referencia `this` de una clase externa, utilice `NombreClaseExterna.this`.
4. La clase externa es responsable de crear objetos de sus clases internas. Para crear un objeto de otra clase interna de la clase, primero genere un objeto de la clase externa y asígnelo a una referencia (a la que llamaremos `ref`). Después utilice una instrucción de la siguiente forma para crear un objeto de clase interna:
`NombreClaseExterna.NombreClaseInterna innerRef = ref.new NombreClaseInterna();`
5. Una clase interna puede declararse como `static`. Una clase interna `static` no requiere que se defina un objeto de su clase externa (mientras que una clase interna no estática sí lo necesita). Una clase interna `static` no tiene acceso a los miembros no estáticos de la clase externa.

27.19 Clases envolventes para tipos primitivos

Cada uno de los tipos primitivos tiene una *clase de tipo envolvente*. A estas clases se les conoce como `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` y `Boolean`. Cada clase de tipo envoltura le permite manipular tipos primitivos como objetos de la clase `Object`. Por lo tanto, los valores de tipos de datos primitivos pueden procesarse de manera polimórfica, si se mantienen como objetos de clases de tipo envoltura. Muchas de las clases que desarrollaremos o que reutilizaremos manipulan y comparten objetos. Estas clases no pueden manipular de manera polimórfica variables de tipos primitivos, pero pueden manipular de manera polimórfica objetos de las clases de tipo envoltura, ya que en última instancia, toda clase se deriva de la clase `Object`.

Cada una de las clases numéricas (`Byte`, `Short`, `Integer`, `Long`, `Float` y `Double`) hereda de la clase `Number`. Cada uno de los tipos de envoltura se declara `final`, por lo que sus métodos son implícitamente `final` y no pueden redefinirse. Observe que muchos de los métodos que procesan los tipos de datos primitivos se definen como métodos `static` de las clases de tipo envoltura. Si necesita manipular un valor primitivo en su programa, primero revise la documentación para las clases de tipo envoltura; es posible que el método que necesita ya esté definido.

RESUMEN

- Una de las claves del poder de la programación orientada a objetos es lograr la reutilización de software a través de la herencia.
- A través de la herencia, una nueva clase hereda las variables y los métodos de instancia de una superclase previamente definida. En este caso, a la nueva clase se le conoce como subclase.
- Con herencia simple, una clase se deriva de una superclase. Con herencia múltiple, una subclase hereda de muchas superclases. Java no soporta la herencia múltiple, pero proporciona la idea de las interfaces, la cual ofrece muchos de los beneficios de la herencia múltiple sin los problemas asociados.

- Una subclase normalmente agrega variables y métodos de instancia por sí misma, por lo que una subclase generalmente es más grande que su superclase. Una subclase es más específica que su superclase, y normalmente representa pocos objetos.
- Una subclase no puede acceder a los miembros **private** de su superclase. Sin embargo, una subclase accede a los miembros **public**, **protected** y de acceso a paquetes de su superclase; la subclase debe estar en el paquete de la superclase para utilizar a los miembros de la superclase con acceso a paquetes.
- La herencia permite la reutilización de software, la cual ahorra tiempo de desarrollo y motiva el uso de software de alta calidad previamente probado y depurado.
- Algún día, la mayoría del software se construirá a partir de componentes reutilizables estandarizados, exactamente de la misma manera en que actualmente se hace la mayoría del hardware.
- Un objeto de una subclase puede tratarse como un objeto de su superclase correspondiente, pero lo contrario no es verdad.
- Una superclase existe en una relación jerárquica con sus subclases.
- Cuando una clase se utiliza con el mecanismo de la herencia, se vuelve una superclase que proporciona atributos y comportamientos a otras clases, o la clase se vuelve una subclase que hereda dichos atributos y comportamientos.
- Una jerarquía de herencia puede ser arbitrariamente profunda dentro de las limitaciones físicas de un sistema en particular, pero la mayoría de las jerarquías de herencia tienen sólo unos cuantos niveles.
- Las jerarquías son útiles para comprender y manejar la complejidad del software. Debido a que el software se vuelve cada vez más complejo, Java proporciona mecanismos para soportar estructuras jerárquicas a través de la herencia y el polimorfismo.
- El acceso **protected** sirve como un nivel intermedio de protección entre el acceso **public** y el **private**. A los miembros **protected** de una superclase pueden acceder los métodos de la superclase, los métodos de las subclases y los métodos de las clases en el mismo paquete; ningún otro método puede acceder a los miembros **protected** de una superclase.
- Una superclase puede ser una superclase directa de una subclase, o una superclase indirecta de una subclase. Una superclase directa es la clase que una subclase explícitamente amplía por medio de **extends**. Una superclase indirecta hereda de muchos niveles superiores en el árbol de jerarquía de clase.
- Cuando un miembro de una superclase es inadecuado para una subclase, el programador puede redefinir ese miembro en la subclase.
- Es importante diferenciar entre las relaciones *es un* y *tiene un*. En una relación *tiene un*, un objeto de una clase tiene como miembro a una referencia hacia un objeto de otra clase. En una relación *es un*, un objeto de un tipo de subclase también puede tratarse como un objeto de un tipo de superclase. *Es un* es herencia. *Tiene un* es composición.
- Un objeto de una subclase puede asignarse a una referencia de una superclase. Este tipo de asignación tiene sentido debido a que la subclase tiene miembros que corresponden a cada miembro de la superclase.
- Una referencia a un objeto de una subclase puede convertirse implícitamente en una referencia para un objeto de una superclase.
- Es posible convertir una referencia de una superclase en una referencia de una subclase por medio de una conversión de tipo explícita. Si el objetivo no es un objeto de una subclase, se lanza una **ClassCastException**.
- Una superclase especifica similitudes. Todas las clases derivadas de una superclase heredan las capacidades de esa superclase. En el proceso de diseño orientado a objetos, el diseñador busca similitudes entre clases y factores que toma para formar superclases. Las subclases entonces se personalizan más allá de las capacidades heredadas de la superclase.
- Leer un conjunto de declaraciones de subclases puede resultar confuso, ya que los miembros heredados de una superclase no se listan en las declaraciones de la subclase, pero estos miembros están realmente presentes en las subclases.
- Con el polimorfismo, se vuelve posible diseñar e implementar sistemas que son más fácilmente extensibles. Los programas pueden escribirse para procesar objetos de tipos que pueden no existir cuando el programa está en desarrollo.
- La programación polimórfica puede eliminar la necesidad de la lógica de **switch**, con lo que se evitan los tipos de errores asociados con dicha lógica.
- Un método abstracto se declara en la superclase precediendo la definición del método con la palabra reservada **abstract**.
- Existen muchas situaciones en las que es útil definir clases para las que el programador nunca intenta instanciar objeto alguno. Tales clases se conocen como clases **abstract**. Éstas se utilizan sólo como superclases, por lo que normalmente nos referiremos a ellas como superclases **abstract**. Ningún objeto de una clase **abstract** puede instanciarse.
- A las clases cuyos objetos pueden instanciarse se les conoce como clases concretas.

- Una clase se hace abstracta declarándola con la palabra reservada **abstract**.
- Si una subclase se deriva de una superclase con un método **abstract** sin proporcionar una definición para ese método **abstract** en la subclase, ese método permanece como **abstract** en la subclase. Como consecuencia, la subclase también es una clase **abstract** (y no puede instanciar objeto alguno).
- Cuando se hace una solicitud a través de una referencia de superclase para utilizar un método, Java elige el método redefinido correcto en la subclase asociada con el objeto.
- A través del polimorfismo, una llamada a un método puede ocasionar diferentes acciones, de acuerdo con el tipo del objeto que recibe la llamada.
- Aunque no podemos crear instancias de objetos de superclases **abstract**, podemos declarar referencias hacia superclases **abstract**. Tales referencias pueden entonces utilizarse para permitir manipulaciones polimórficas de objetos de subclases, cuando dichos objetos son instanciados desde clases concretas.
- Con regularidad se agregan nuevas clases a los sistemas. Las nuevas clases se acomodan por medio del método de vinculación dinámica (también conocido como vinculación tardía). El tipo de un objeto no necesita conocerse en tiempo de compilación, para que se compile una llamada a un método. En tiempo de ejecución, se selecciona el método apropiado para recibir al objeto.
- Con el método de vinculación dinámica, en tiempo de ejecución, la llamada a un método se envía hacia la versión adecuada del método para la clase del objeto que recibe la llamada.
- Cuando una superclase proporciona un método, las subclases pueden redefinir el método, pero no tienen que hacerlo. Entonces, una subclase puede utilizar una versión de superclase de un método.
- Una definición de interfaz comienza con la palabra reservada **interface**, y contiene un conjunto de métodos **public abstract**. Las interfaces también pueden contener datos **public final static**.
- Para utilizar una interfaz, debe especificarse una clase que la implemente, y esa clase debe definir cada método en la interfaz con el número de argumentos y el tipo de retorno especificado en la definición de la interfaz.
- Por lo general, una interfaz se utiliza en lugar de una clase **abstract**, cuando no existe una implementación predeterminada a heredar.
- Cuando una clase implementa una interfaz, aplica la misma relación *es un* provista por la herencia.
- Para implementar más de una interfaz, en la definición de la clase simplemente proporcione una lista separada por comas con los nombres de las interfaces, después de la palabra reservada **implements**.
- Las clases internas se definen dentro del alcance de otras clases.
- Una clase interna también puede definirse dentro de un método de una clase. Tales clases internas tienen acceso a los miembros externos de la clase y a las variables locales **final** del método en el que están definidas.
- Las definiciones de clases internas se utilizan principalmente para la manipulación de eventos.
- La clase **JFrame** proporciona los atributos y comportamientos básicos de una ventana; una barra de título y botones para minimizar, maximizar y cerrar la ventana.
- Un objeto de una clase interna tiene una relación especial con el objeto de la clase externa que lo crea. Al objeto de la clase interna se le permite acceder directamente a todas las variables y métodos de instancia del objeto de la clase externa.
- Una clase interna anónima no tiene nombre, por lo que un objeto de una clase interna anónima debe crearse en el punto en el que la clase se define en el programa.
- Una clase interna anónima puede implementar una interfaz o extender una clase.
- El evento generado cuando el usuario hace clic en el cuadro **Close** de la ventana, es un evento de cierre de ventana.
- El método **addWindowListener** registra un oyente del evento ventana. Su argumento debe ser una referencia hacia un objeto que es un **WindowListener** (paquete **java.awt.event**).
- Para interfaces de manejo de eventos con más de un método, Java proporciona una clase correspondiente (llamada clase adaptadora) que implementa para usted todos los métodos en la interfaz. La clase **WindowAdapter** implementa la interfaz **WindowListener**, de tal forma que todo objeto **WindowAdapter** *es un* **WindowListener**.
- Compilar una clase que contiene clases internas da como resultado un archivo **.class** para cada clase.
- Las clases internas con nombres de clases pueden definirse como **public**, **protected**, de acceso a paquetes o **private**, y están sujetas a las mismas restricciones de uso que los otros miembros de una clase.
- Para acceder a la referencia **this** de una clase externa, utilice **NombreClaseExterna.this**.
- La clase externa es responsable de crear objetos de sus clases internas no estáticas.
- Una clase interna puede declararse como **static**.

TERMINOLOGÍA

abstracción	control de acceso a miembros	programación orientada a objetos (POO)
clase abstract	conversión implícita	recolector de basura
clase base	de referencia	redefinición <i>vs</i> sobrecarga
clase Boolean	extends	redefinir un método
clase Character	extensibilidad	redefinir un método abstract
clase Double	herencia	referencia hacia una clase abstract
clase envolvente	herencia de implementación	
clase final	herencia de interfaz	
clase Integer	herencia múltiple	referencia hacia una subclase
clase interna	herencia simple	referencia hacia una superclase
clase interna anónima	interfaz	relación <i>es un</i>
clase JFrame	interfaz WindowListener	relación jerárquica
clase Long	jerarquía de clase	relación <i>tiene un</i>
clase Number	jerarquía de herencia	reutilización de software
clase Object	lógica switch	subclase
clase WindowAdapter	método abstract	super
clase WindowEvent	método de vinculación dinámica	superclase
cliente de una clase	método final	superclase abstract
componentes de software estandarizados	método show	superclase directa
composición	método windowClosing	superclase indirecta
constructor de una subclase	miembro protected de una clase	this
constructor de una superclase	objeto miembro	variable de instancia final
	polimorfismo	vinculación tardía

ERRORES COMUNES DE PROGRAMACIÓN

- 27.1 Tratar a un objeto de una superclase como un objeto de una subclase puede ocasionar errores.
- 27.2 Asignar un objeto de una superclase a una referencia de una subclase (sin una conversión de tipo), es un error de sintaxis.
- 27.3 Si una subclase hace una llamada **super** al constructor de su superclase, y esta llamada no es la primera instrucción en el constructor de la subclase, es un error de sintaxis.
- 27.4 Si los argumentos de una llamada **super** de una subclase al constructor de su superclase no coinciden con los parámetros especificados en una de las definiciones del constructor de la superclase, es un error de sintaxis.
- 27.5 Si un método de una superclase y un método en su subclase tienen la misma firma pero diferente tipo de retorno, es un error de sintaxis.
- 27.6 Asignar un objeto de subclase a una referencia de superclase, y después intentar hacer referencia sólo a miembros de la subclase con la referencia de superclase, es un error de sintaxis.
- 27.7 Intentar crear una instancia de un objeto de una clase abstracta (es decir, una clase que contiene uno o más métodos abstractos), es un error de sintaxis.
- 27.8 Si una clase con uno o más métodos **abstract** no se declara específicamente como **abstract**, es un error de sintaxis.
- 27.9 Dejar indefinido un método de una interfaz, en una clase que implementa la interfaz, da como resultado un error de compilación que indica que la clase debe declararse como **abstract**.
- 27.10 Extender una clase adaptadora y escribir incorrectamente el nombre de un método que va a redefinir, es un error de lógica.

TIPS PARA PREVENIR ERRORES

- 27.1 Ocultar los miembros **private** es una gran ayuda al probar, depurar y modificar correctamente los sistemas. Si una subclase pudiera acceder a los miembros **private** de su superclase, entonces sería posible que las clases derivadas de esa subclase accedieran también a esos datos, y así sucesivamente. Esto propagaría el acceso a lo que se supone deberían ser datos **private**, y los beneficios del ocultamiento de información se perderían a lo largo de la jerarquía de la clase.

- 27.2** Una consecuencia interesante de utilizar el polimorfismo es que los programas adquieren una apariencia simplificada; contienen menos lógica de separación, a favor de un código secuencial más sencillo. Esta simplificación facilita el probar, depurar y mantener un programa.

TIPS DE RENDIMIENTO

- 27.1** Si las clases producidas a través de la herencia son más grandes de lo necesario, podrían desperdiciarse recursos de memoria y de procesamiento. Herede de la clase “que más se acerque” a lo que usted necesita.
- 27.2** El compilador puede decidir poner en línea a una llamada a un método **final**, y lo hará para métodos **final** pequeños y sencillos. Colocarlas en línea no viola el encapsulamiento o el ocultamiento de información (pero mejora el rendimiento, ya que elimina la sobrecarga de realizar una llamada a un método).
- 27.3** Los preprocesadores canalizados pueden mejorar el rendimiento ejecutando simultáneamente diversas partes de las siguientes instrucciones, pero no si esas instrucciones siguen a una llamada a un método. Colocar en línea al código (lo que el compilador realiza en un método **final**) puede mejorar el rendimiento de estos preprocesadores, ya que elimina la transferencia de control fuera de línea asociada con una llamada a un método.
- 27.4** Cuando el polimorfismo se implementa con el método de vinculación dinámica, es eficiente.
- 27.5** Los tipos de manipulaciones polimórficas que se hacen posibles con la vinculación dinámica, también pueden lograrse por medio de la lógica de **switch** codificada manualmente, de acuerdo con los campos de tipo de los objetos. El código polimórfico generado por el compilador de Java se ejecuta con un rendimiento comparable con la lógica de **switch** eficientemente codificada.

OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 27.1** Una subclase no puede acceder directamente a miembros **private** de su superclase.
- 27.2** Los constructores nunca se heredan; éstos son específicos de la clase en la que están definidos.
- 27.3** Si un objeto se ha asignado a una referencia de una de sus superclases, es aceptable convertir el tipo de ese objeto de regreso a su propio tipo. De hecho, esto debe hacerse para enviar a ese objeto cualquiera de los mensajes que no aparecen en esa superclase.
- 27.4** Toda clase en Java extiende a **Object**, a menos que se especifique lo contrario en la primera línea de la definición de la clase. Por lo tanto, la clase **Object** es la superclase de toda la jerarquía de clases de Java.
- 27.5** Una redefinición de un método de una superclase en una subclase no tiene la misma firma que el método de la superclase. Tal redefinición no es la redefinición de un método, sino un simple ejemplo de la sobrecarga de métodos.
- 27.6** Cualquier objeto puede convertirse en una **String** con una llamada explícita o implícita al método **toString** del objeto.
- 27.7** Toda clase debe redefinir el método **toString** para devolver información útil sobre los objetos de esa clase.
- 27.8** Crear una subclase no afecta el código fuente de su superclase, o el código en bytes de las superclases de Java; la integridad de una superclase se preserva a través de la herencia.
- 27.9** Así como el diseñador de sistemas no orientados a objetos deben evitar la proliferación de funciones innecesarias, el diseñador de sistemas orientados a objetos debe evitar la proliferación de clases innecesarias. La proliferación de clases genera problemas de administración y puede dificultar la reutilización de software, simplemente porque es más difícil para un usuario potencial de una clase localizar esa clase en una amplia colección. El equilibrio se encuentra en crear pocas clases que proporcionen funcionalidad adicional importante, sin embargo, dichas clases pueden ser demasiado ricas para ciertos usuarios.
- 27.10** En un sistema orientado a objetos, con frecuencia las clases se encuentran muy relacionadas. “Ubique” los atributos y comportamientos comunes y colóquelos en una superclase. Después utilice la herencia para formar subclases para que no tenga que repetir atributos y comportamientos comunes.
- 27.11** Las modificaciones a una superclase no requieren que las subclases se modifiquen, mientras la interfaz pública de la superclase permanezca sin cambios.
- 27.12** Cuando una subclase elige no redefinir un método, la subclase simplemente hereda la definición del método de su superclase inmediata.
- 27.13** Una clase definida como **final** no puede extenderse, y cada uno de sus métodos es implícitamente **final**.
- 27.14** Una clase abstracta puede tener datos de instancia y métodos no abstractos sujetos a las reglas normales de la herencia de las subclases. Una clase abstracta también pueden tener constructores.

- 27.15** Si una subclase se deriva de una superclase con un método **abstract**, y si no se proporciona una definición en la subclase para ese método **abstract** (es decir, si no se redefine ese método en la subclase), ese método permanece como **abstract** en la subclase. Como consecuencia, la subclase también es una clase **abstract**, y debe declararse explícitamente como **abstract**.
- 27.16** La habilidad de declarar un método **abstract** le da al diseñador de la clase suficiente poder sobre cómo implementará las subclases en una jerarquía de clases. Cualquier clase nueva que quiera heredar de esta clase es forzada a redefinir el método **abstract** (ya sea directamente o heredando de una clase que ha redefinido el método). De lo contrario, esa nueva clase contendrá un método **abstract** y, por lo tanto, será una clase **abstract** incapaz de instanciar objetos.
- 27.17** Con el polimorfismo, el programador puede lidiar con las generalidades y dejar que el ambiente en tiempo de ejecución se ocupe de lo específico. El programador puede ordenar que una amplia variedad de objetos se comporten de manera apropiada sin siquiera conocer los tipos de esos objetos.
- 27.18** El polimorfismo promueve la extensibilidad: El software escrito para invocar un comportamiento polimórfico se escribe de manera independiente a los tipos de los objetos a los que se envían los mensajes (es decir, llamadas a métodos). Por lo tanto, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en tales sistemas sin modificar el sistema base.
- 27.19** Si un método se declara como **final**, éste no puede redefinirse en las subclases, por lo que las llamadas al método no pueden enviarse de manera polimórfica a los objetos de esas subclases. La llamada al método aún puede enviarse a las subclases, pero responderán de manera idéntica, en lugar de hacerlo de manera polimórfica.
- 27.20** Una clase **abstract** define una interfaz común para los diversos miembros de una jerarquía de clase. La clase **abstract** contiene métodos que se definirán en las subclases. Todas las clases de la jerarquía pueden utilizar esta misma interfaz a través del polimorfismo.
- 27.21** Las jerarquías diseñadas para la herencia de la implementación tienden a tener a su funcionalidad arriba en la jerarquía; cada nueva subclase hereda uno o más de los métodos que se definieron en una superclase, y utiliza las definiciones de la superclase.
- 27.22** Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad más abajo en la jerarquía; una superclase especifica uno o más métodos que deben invocarse de manera idéntica para cada objeto en la jerarquía (es decir, tienen la misma firma), pero las subclases individuales proporcionan sus propias implementaciones de los métodos.
- 27.23** Una subclase siempre hereda la versión definida más recientemente de cada método **public** y **protected** de sus superclases directa e indirecta.
- 27.24** Todos los métodos de la clase **Object** pueden invocarse por medio de una referencia a un tipo de dato interfaz; una referencia se refiere a un objeto, y todos los objetos tienen los métodos definidos por la clase **Object**.
- 27.25** A un objeto de la clase interna se le permite tener acceso directo a las variables de instancia y a los métodos del objeto de la clase externa que la define.
- 27.26** Una clase interna definida dentro de un método tiene acceso directo a todas las variables de instancia y métodos del objeto de la clase externa en la que se define y en cualquier variable local de **final** en el método.
- 27.27** Cuando una clase anónima interna implementa una interfaz, la clase debe definir cada método en la interfaz.

EJERCICIOS DE AUTOEVALUACIÓN

- 27.1** Complete los espacios en blanco:
- Si la clase **Alfa** hereda de la clase **Beta**, a la clase **Alfa** se le conoce como _____, y a la clase **Beta** se le conoce como _____.
 - La herencia permite la _____, la cual ahorra tiempo de desarrollo y motiva el uso de componentes de software de alta calidad previamente probados.
 - Un objeto de una clase puede tratarse como un objeto de su _____ correspondiente.
 - Los cuatro especificadores de acceso a miembros son _____, _____, _____ y _____.
 - Una relación *tiene un* entre clases representa a la _____, y una relación *es un* entre clases representa a la _____.
 - Utilizar el polimorfismo ayuda a eliminar la lógica de _____.
 - Si una clase contiene uno o más métodos **abstract**, se trata de una clase _____.
 - Una llamada a un método resuelta en tiempo de ejecución se conoce como vinculación _____.

- 27.2** a) Una subclase puede llamar a cualquier método de una superclase no **private**, anteponiendo _____ a la llamada al método.
 b) Una superclase generalmente representa a un número mayor de objetos que su subclase. (*Verdadero/falso.*)
 c) Una subclase normalmente encapsula menos funcionalidad que su superclase. (*Verdadero/falso.*)

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

- 27.1** a) Subclase, superclase. b) Reutilización de software. c) Subclase, superclase. d) **public**, **protected**, **private** y de acceso a paquetes. e) Composición, herencia. f) **switch**. g) **abstract**. h) Dinámica.
- 27.2** a) **super**
 b) verdadero
 c) falso

EJERCICIOS

- 27.3** Considere la clase **Bicicleta**. Dado su conocimiento sobre algunos componentes de bicicletas, muestre una jerarquía en la que la clase **Bicicleta** herede de otras clases, las cuales, a su vez, hereden de otras clases. Explique la creación de instancias de varios objetos de la clase **Bicicleta**. Explique la herencia de la clase **Bicicleta** para otras subclases muy relacionadas.
- 27.4** Defina cada uno de los siguientes términos: herencia simple, herencia múltiple, interfaz, superclase y subclase.
- 27.5** Explique por qué convertir el tipo de una referencia de superclase en una referencia de subclase es potencialmente peligroso.
- 27.6** Plantee las diferencias entre la herencia simple y la herencia múltiple. ¿Por qué Java no soporta la herencia múltiple? ¿Qué característica de Java ayudan a contar con los beneficios de la herencia múltiple?
- 27.7** (*Verdadero/Falso.*) Una subclase es generalmente más pequeña que su superclase.
- 27.8** (*Verdadero/Falso.*) Un objeto de una subclase es también un objeto de la superclase de esa subclase.
- 27.9** Rescriba el programa **Punto**, **Círculo**, **Cilindro** de la figura 27.4 como un programa **Punto**, **Cuadrado**, **Cubo**. Haga esto de dos formas: una con herencia y otra con composición.
- 27.10** En el capítulo dijimos que “cuando un método de una superclase es inadecuado para una subclase, ese método puede redefinirse en la subclase con una implementación adecuada”. Si se hace esto, ¿la relación “el objeto de una subclase es un objeto de la superclase”, se mantiene? Explique su respuesta.
- 27.11** ¿Cómo es que el polimorfismo le permite programar “en general”, en lugar de “en específico”? Explique las principales ventajas de la programación “en general”.
- 27.12** Explique los problemas de la programación con lógica de **switch**. Explique por qué el polimorfismo es una alternativa efectiva al uso de la lógica de **switch**.
- 27.13** Plantee la diferencia entre herencia de interfaz y herencia de implementación. ¿Cómo difieren las jerarquías de herencia diseñadas para herencia de interfaz de aquellas diseñadas para herencia de implementación?
- 27.14** Plante la diferencia entre métodos no abstractos y los métodos abstractos.
- 27.15** (*Verdadero/Falso.*) Todos los métodos de una superclase **abstract** deben declararse como **abstract**.
- 27.16** Sugiera uno o más niveles de superclases **abstract** para la jerarquía **Figura** que explicamos al principio de este capítulo (el primer nivel es **Figura**, y el segundo nivel consiste en las clases **FiguraBidimensional** y **FiguraTridimensional**).
- 27.17** ¿Cómo es que el polimorfismo promueve la extensibilidad?
- 27.18** Se le ha pedido que desarrolle un simulador de vuelo que tendrá que elaborar resultados gráficos. Explique por qué la programación polimórfica sería especialmente efectiva para un problema de esta naturaleza.
- 27.19** (*Aplicación de dibujo.*) Modifique el programa de dibujo del ejercicio 26.11 para crear una aplicación de dibujo que dibuje líneas aleatorias, rectángulos y óvalos. [Nota: Como un applet, **JFrame** tiene un método **paint** que puede redefinir para dibujar en el fondo del **JFrame**.]

Para este ejercicio, modifique las clases **MiLinea**, **MiElipse** y **MiRectangulo** del ejercicio 26.11 para crear la jerarquía de clase de la figura 27.8. Las clases de la jerarquía **MiFigura** deben ser clases de figuras “inteligentes”, en donde los objetos de estas clases sepan cómo dibujarse a sí mismas (si cuentan con un objeto **Graphics** que les indique dónde dibujar). La única lógica de **switch** o de **if/else** en este programa debe ser

para determinar el tipo de objeto figura a crear (utilice números aleatorios para escoger el tipo de figura y las coordenadas de cada figura.) Una vez que se cree un objeto de esta jerarquía, éste será manipulado por el resto de su tiempo de vida como una referencia de la superclase **MiFigura**.

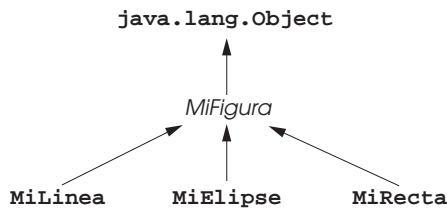


Figura 27.8 Jerarquía **MiFigura**.

La clase **MiFigura** de la figura 27.8 *debe ser abstract*. El único dato que representa las coordenadas de las figuras de la jerarquía debe definirse en la clase **MiFigura**. Las líneas, los rectángulos y las elipses pueden dibujarse si conoce dos puntos en el espacio. Las líneas requieren coordenadas *x1*, *y1* y *x2*, *y2*. El método `drawLine` de la clase **Graphics** conectará con una línea los dos puntos proporcionados. Si usted tiene los mismos cuatro valores para las coordenadas (*x1*, *y1* y *x2*, *y2*) para elipses y rectángulos, puede calcular los cuatro argumentos necesarios para dibujarlos. Cada uno requiere un valor para la coordenada superior izquierda *x* (el mínimo de los dos valores para las coordenadas en *x*), un valor para la coordenada superior izquierda *y* (el mínimo de los dos valores para las coordenadas en *y*), un *ancho* (la diferencia entre los dos valores correspondientes a las coordenadas en *x*; la cual debe ser positiva) y una *altura* (la diferencia entre los dos valores correspondientes a las coordenadas en *y*; la cual debe ser positiva). [Nota: En el capítulo 29, cada par *x,y* se capturará utilizando eventos del ratón, a partir de interacciones del ratón entre el usuario y el fondo del programa. Estas coordenadas se almacenarán en el objeto de figura adecuado, conforme seleccione el usuario. Conforme inicie el ejercicio, utilizará valores aleatorios para las coordenadas como argumentos del constructor.]

Además de los datos para la jerarquía, la clase **MiFigura** debe definir al menos los siguientes métodos:

- Un constructor sin argumentos que establezca en cero a las coordenadas.
- Un constructor con argumentos que establezca las coordenadas en los valores proporcionados.
- Métodos *establecer* para cada pieza individual de datos que permita al programador establecer de manera independiente cualquier pieza de datos para una figura de la jerarquía (por ejemplo, si tiene una variable de instancia **x1**, debe tener un método **estableceX1**).
- Métodos *obtener* para cada pieza individual de datos que permita al programador recuperar de manera independiente cualquier pieza de datos para una figura de la jerarquía (por ejemplo, si tiene una variable de instancia **x1**, debe tener un método **obtienex1**).
- El método **abstract**

```
public abstract void draw( Graphics g );
```

Este método será llamado desde el método **paint** del programa para dibujar una figura en la pantalla.

Los métodos anteriores son necesarios. Si quisiera proporcionar más métodos para una mayor flexibilidad, hágalos. Sin embargo, asegúrese de que cualquier método que defina en esta clase sea un método que se utilizará en *todas* las figuras de la jerarquía.

Todos los datos deben ser **private** para la clase **MiFigura** de este ejercicio (esto lo obliga a utilizar el encapsulamiento de datos adecuado, y a proporcionar los métodos *establecer*/*obtener* adecuados para manipular los datos). No se le permite definir nuevos datos que puedan derivarse de información existente. Como explicamos anteriormente, la *x* superior izquierda, la *y* superior izquierda, el *ancho* y la *altura* son necesarios para dibujar un óvalo o para calcular un rectángulo, si usted ya conoce dos puntos en el espacio. Todas las subclases de **MiFigura** deben proporcionar dos constructores que imiten a los proporcionados por la clase **MiFigura**.

Los objetos de las clases **MiElipse** y **MiRecta** no deben calcular sus coordenadas superiores izquierdas *x* y *y*, y el *ancho* y la *altura*, hasta que se vayan a dibujar. Nunca modifique las coordenadas *x1*, *y1* y *x2* y *y2* de un objeto **MiElipse** o **MiRecta** para prepararse a dibujarlos. En su lugar, utilice resultados temporales de los cálculos descritos arriba. Esto nos ayudará a mejorar el programa del capítulo 29, que permitirá al usuario seleccionar con el ratón las coordenadas de cada figura.

En el programa no debe haber referencias **MiLinea**, **MiElipse** o **MiRecta**; sólo están permitidas las referencias de **MiFigura** que hagan referencia a objetos **MiLinea**, **MiElipse** y **MiRecta**.

El programa debe mantener un arreglo de referencias **MiFigura** que contenga todas las figuras. El método **paint** del programa deben recorrer el arreglo de referencias **MiFigura** y dibujar todas las figuras (es decir, llamar a cada método **draw** de las figuras).

Comience definiendo la clase **MiFigura**, la clase **MiLinea** y una aplicación para probar sus clases. La aplicación debe tener una variable de instancia que pueda referirse a un objeto **MiLinea** (creado en el constructor de la aplicación). El método **paint** (para su subclase **JFrame**) debe dibujar la figura con una instrucción como

```
figuraActual.draw( g );
```

donde **figuraActual** es la referencia **MiFigura** y **g** es el objeto **Graphics** que la figura utilizará para dibujarse a sí misma en el fondo de la ventana.

Después, cambie la referencia simple **MiFigura** hacia un arreglo de referencias de **MiFigura**, y codifique diversos objetos **MiLinea** en el programa de dibujo. El método **paint** de la aplicación debe recorrer el arreglo de figuras y dibujar cada figura.

Después de que la parte anterior esté funcionando, debe definir las clases **MiElipse** y **MiRecta**, y agregar objetos de estas clases en el arreglo existente. Por el momento, todos los objetos de figuras deben crearse en el constructor de su subclase **JFrame**. En el capítulo 29, crearemos los objetos cuando el usuario elija una figura y comience a dibujarlo con el ratón.

28

Gráficos en Java y Java2D

Objetivos

- Comprender los contextos y los objetos gráficos.
- Comprender y manipular colores.
- Comprender y manipular fuentes.
- Comprender y utilizar los métodos de **Graphics** para dibujar líneas, rectángulos, rectángulos con esquinas redondeadas, rectángulos de tres dimensiones, elipses, arcos y polígonos.
- Utilizar los métodos de la clase **Graphics2D** de la API **Java2D** para dibujar líneas, rectángulos, rectángulos con líneas redondeadas, elipses, arcos y patrones generales.
- Especificar las características **Paint** y **Stroke** de las figuras desplegadas con **Graphics2D**.

Una imagen vale más que mil palabras.

Proverbio Chino

*Trata a la naturaleza como a un cilindro, una esfera, un cono,
todas en perspectiva.*

Paul Cézanne

*Nada es real hasta que se experimenta, incluso un proverbio
no es proverbio hasta que la vida se lo ilustra.*

John Keats

*Una imagen me muestra al instante lo que a un libro le lleva
docenas de páginas.*

Ivan Sergeyevich



Plan general

- 28.1 Introducción**
- 28.2 Contextos gráficos y objetos gráficos**
- 28.3 Control del color**
- 28.4 Control de fuentes**
- 28.5 Cómo dibujar líneas, rectángulos y elipses**
- 28.6 Cómo dibujar arcos**
- 28.7 Cómo dibujar polígonos y polilíneas**
- 28.8 La API Java2D**
- 28.9 Figuras en Java2D**

Resumen • Terminología • Errores comunes de programación • Tips de portabilidad • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

28.1 Introducción

En este capítulo, echaremos un vistazo a las capacidades de Java para dibujar figuras de dos dimensiones, para controlar los colores y para controlar las fuentes. Uno de los atractivos iniciales de Java era el soporte para gráficos que permitía a los programadores mejorar visualmente sus applets y aplicaciones. Ahora, Java contiene muchas más capacidades sofisticadas que forman parte de la API *Java2D*. Este capítulo comienza con una introducción a muchas de las capacidades originales de Java. A continuación, presentamos varias de las nuevas y más poderosas capacidades de Java2D, tales como el control del estilo de las líneas que se utilizan para dibujar las figuras y el control para rellenar figuras con colores y patrones.

La figura 28.1 muestra una parte de la jerarquía de clases de Java que incluyen varias de las distintas clases para gráficos básicos y las clases de la API Java2D, así como las interfaces que hemos tratado en este libro. La clase **Color** contiene los métodos y las constantes para manipular colores. La clase **Font** contiene los métodos y las constantes para manipular fuentes. La clase **FontMetrics** contiene los métodos obtener la información de las fuentes. La clase **Polygon** contiene los métodos para crear polígonos. La clase **Graphics** contiene los métodos para dibujar cadenas, líneas, rectángulos y otras figuras. La mitad inferior de la figura lista varias clases e interfaces de la API Java2D. La clase **BasicStroke** ayuda a especificar las características de las líneas. Las clases **GradientPaint** y **TexturePaint** ayudan a especificar las características para el relleno de las figuras con colores y patrones. Las clases **GeneralPath**, **Arc2D**, **Ellipse2D**, **Line2D**, **Rectangle2D**, y **RoundRectangle2D** definen una variedad de figuras de **Java2D**.

Para comenzar a dibujar en Java, primero debemos comprender el *sistema de coordenadas* de Java (figura 28.2), el cual es un esquema para identificar cada posible punto en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente GUI (tal como un applet o una ventana) tiene las coordenadas (0,0). Un par de coordenadas está compuesto por una *coordenada x* (la *coordenada horizontal*) y una *coordenada y* (la *coordenada vertical*). La coordenada *x* es la distancia horizontal de movimiento hacia la derecha, desde la esquina superior izquierda. La coordenada *y* es la distancia vertical de movimiento hacia abajo, desde la esquina superior izquierda. El *eje x* describe cada coordenada horizontal, y el *eje y* describe cada coordenada vertical.

Observación de ingeniería de software 28.1



La coordenada superior izquierda (0,0) de una ventana en realidad se encuentra debajo de la barra de título de la ventana. Por esta razón, las coordenadas de dibujo deben ajustarse para dibujar dentro de los bordes de la ventana. La clase **Container** (una superclase de todas las ventanas en Java) contiene el método **getInsets** que devuelve un objeto **Inset** (del paquete **java.awt**) para este propósito. Un objeto **Inset** contiene cuatro miembros públicos, **top**, **bottom**, **left** y **right**, que representan el número de píxeles de cada borde de la ventana hacia el área de dibujo de ésta.

El texto y las figuras se despliegan en la pantalla especificando las coordenadas. Las unidades de las coordenadas se miden en *píxeles*. Un *píxel* es la unidad de resolución más pequeña de la pantalla.

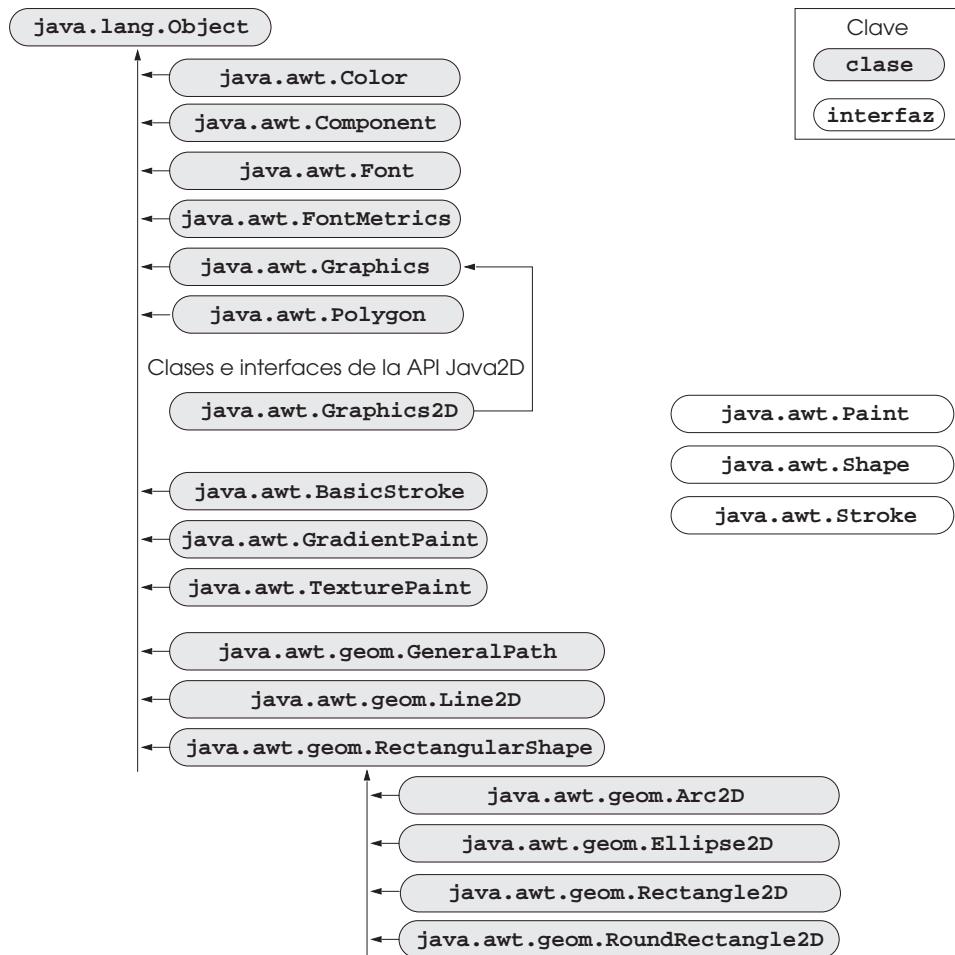


Figura 28.1 Algunas clases e interfaces de las capacidades gráficas originales de Java y de la API Java2D, que utilizamos en este capítulo.

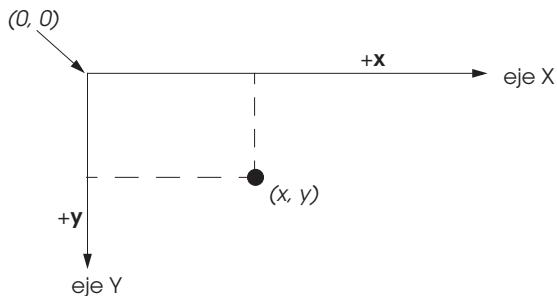


Figura 28.2 Sistema de coordenadas de Java. Las unidades se miden en pixeles.



Tip de portabilidad 28.1

Diferentes pantallas tienen diferentes resoluciones (es decir, varía la densidad de pixeles). Esto puede provocar que los gráficos parezcan de tamaño diferente en diferentes pantallas.

28.2 Contextos gráficos y objetos gráficos

Un *contexto gráfico* en Java permite dibujar en la pantalla. Un objeto de **Graphics** manipula un contexto gráfico al controlar la forma en que se dibuja en él. Los objetos de **Graphics** contienen métodos para dibujar, para manipular fuentes, para manipular colores y otros aspectos similares. Cada uno de los applets que vimos en el texto que realiza el dibujo en la pantalla utiliza en el objeto **g** de **Graphics** (el argumento para el método **paint** del applet) para manipular el contexto gráfico del applet. En este capítulo, mostraremos las aplicaciones de dibujo. Sin embargo, cada técnica mostrada puede ser útil en los applets.

La clase **Graphics** es una clase **abstract** (es decir, los objetos de **Graphics** no pueden instanciarse). Esto contribuye a la portabilidad de Java. El dibujo se realiza de manera diferente en cada plataforma que soporta Java, de modo que no puede existir una clase que implemente todas las capacidades de dibujo en un solo sistema. Por ejemplo, las capacidades gráficas que permiten a una PC que ejecuta Microsoft Windows dibujar un rectángulo, son diferentes de las capacidades que permiten a una estación de trabajo en UNIX dibujar el mismo rectángulo, y ambas son diferentes a las capacidades que permiten a una Macintosh dibujar un rectángulo. Cuando se implementa Java en cada plataforma, se crea una clase derivada de **Graphics** que en realidad implementa todas las capacidades de dibujo. Esta implementación se nos oculta por medio de la clase **Graphics**, la cual suministra la interfaz que nos permite escribir programas para utilizar gráficos de manera independiente de la plataforma.

La clase **Component** es la superclase de muchas de las clases en el paquete **java.awt** (explicaremos la clase **Component** en el capítulo 29). El método **paint** de **Component** toma un objeto **Graphics** como argumento. Este objeto se pasa al método **paint** mediante el sistema, cuando se requiere una operación **paint** para un **Componente**. El encabezado para el método **paint** es

```
public void paint( Graphics g )
```

El objeto **paint** recibe una referencia a un objeto de la clase derivada de **Graphics**. El método anterior debe parecerle conocido; es el mismo que hemos utilizado en nuestras clases de applets. En realidad, la clase **Component** es una clase base indirecta de la clase **JApplet**, la superclase de cada applet del libro. El método **paint** definido en la clase **Component** no hace cosa alguna de manera predeterminada, el programador la debe redefinir.

Por lo general, el programador llama directamente al método **paint**, debido a que dibujar los gráficos es un *proceso controlado por eventos*. Cuando se ejecuta un applet, al método **paint** se le llama automáticamente (después de las llamadas a los métodos **init** y **start**). Para poder llamar de nuevo a **paint**, debe ocurrir un *evento* (tal como cubrir o descubrir un applet). De manera similar, cuando se despliega un **Component**, se llama al método **paint** de dicho componente.

Si el programador necesita llamar a **paint**, se hace una llamada al método **repaint** de la clase **paint**. El método **repaint** solicita al usuario una llamada al método **update** de la clase **Component** tan pronto como sea posible limpiar cualquier dibujo previo, del fondo del componente, luego **update** llama directamente a **paint**. El programador llama con frecuencia al método **repaint** para forzar la operación **paint**. El método **repaint** no debe redefinirse debido a que realiza algunas tareas dependientes del sistema. Con frecuencia, al método **update** se le llama de manera directa y algunas veces se redefine. Redefinir el método **update** es útil para “suavizar” las animaciones (es decir, reducir las “asperezas”) como explicaremos en el capítulo 30. Los encabezados para **repaint** y **update** son

```
public void repaint()
public void update( Graphics g )
```

El método **update** toma un objeto **Graphics** como argumento, el cual es suministrado automáticamente por el sistema cuando se llama a **update**.

En este capítulo, nos concentraremos en el método **paint**. En el siguiente capítulo nos concentraremos más en la naturaleza controlada por eventos de los gráficos y explicaremos los métodos **repaint** y **update** con más detalle. También explicaremos la clase **JComponent**, una superclase de muchos componentes GUI en el paquete **javax.swing**. Por lo general, las subclases de **JComponent** pintan a partir de los métodos **paint** de **Component**.

28.3 Control del color

Los colores mejoran la apariencia de un programa y ayudan a trasmitir su significado. Por ejemplo, una luz de semáforo tiene tres diferentes luces de colores, el rojo indica alto, el amarillo indica precaución y el verde indica adelante.

La clase **Color** define los métodos y las constantes para manipular los colores en un programa en Java. Las constantes para los colores predefinidos aparecen en la figura 28.3, y la figura 28.4 resume distintos métodos y constructores de colores. Observe que los dos métodos de la figura 28.4 son los métodos de **Graphics** que son específicos para los colores.

Constante del color	Color	valor RGB (RVA)
<code>public final static Color orange</code>	naranja	255, 200, 0
<code>public final static Color pink</code>	rosa	255, 175, 175
<code>public final static Color cyan</code>	cian	0, 255, 255
<code>public final static Color magenta</code>	magenta	255, 0, 255
<code>public final static Color yellow</code>	amarillo	255, 255, 0
<code>public final static Color black</code>	negro	0, 0, 0
<code>public final static Color white</code>	blanco	255, 255, 255
<code>public final static Color gray</code>	gris	128, 128, 128
<code>public final static Color lightGray</code>	gris claro	192, 192, 192
<code>public final static Color darkGray</code>	gris oscuro	64, 64, 64
<code>public final static Color red</code>	rojo	255, 0, 0
<code>public final static Color green</code>	verde	0, 255, 0
<code>public final static Color blue</code>	azul	0, 0, 255

Figura 28.3 Constantes estáticas de la clase **Color** y valores RGB (RVA).

Método	Descripción
<code>public Color(int r, int g, int b)</code>	Crea un color basado en contenido de rojo, verde y azul expresados como enteros desde 0 hasta 255.
<code>public Color(float r, float g, float b)</code>	Crea un color basado en contenido de rojo, verde y azul expresados como flotantes entre 0.0. y 1.0.
<code>public int getRed() // clase Color</code>	Devuelve un valor entre 0 y 255 que representa el contenido de rojo.
<code>public int getGreen() // clase Color</code>	Devuelve un valor entre 0 y 255 que representa el contenido de verde.
<code>public int getBlue() // clase Color</code>	Devuelve un valor entre 0 y 255 que representa el contenido de azul.
<code>public Color getColor() // clase Graphics</code>	Devuelve un objeto Color que representa el color actual para el contexto gráfico.
<code>public void setColor(Color c) // clase Graphics</code>	Establece el color actual para dibujo con el contexto gráfico.

Figura 28.4 Los métodos **Color** y los métodos relacionados con el color de **Graphics**.

Todos los colores se crean a partir de los componentes de rojo, verde y azul. Estos componentes se llaman *valores RGB* (RVA). Los tres componentes RGB pueden ser enteros en el rango de 0 a 255, o pueden ser valores en punto flotante en el rango de 0.0 a 1.0. La primera parte de RGB define la cantidad de rojo, la segunda define la cantidad de verde y la tercera define la cantidad de azul. El valor RGB más grande será la cantidad de un color en particular. Java permite al programador elegir de entre $256 \times 256 \times 256$ (o aproximadamente 16.7 millones) de colores. Sin embargo, no todas las computadoras son capaces de desplegar todos estos colores. Si éste es el caso, la computadora desplegará el color más cercano posible.

Error común de programación 28.1



*Escribir cualquier constante estática de clase de **Color** con una letra mayúscula inicial, es un error de sintaxis.*

En la figura 28.4 aparecen dos constructores **Color**, uno que toma tres argumentos **int** y uno que toma tres argumentos **float**, en donde cada argumento especifica la cantidad de rojo, verde, y azul, respectivamente. Los valores **int** deben estar entre 0 y 255, y los valores **float** deben estar entre 0.0 y 1.0. El nuevo objeto **Color** contendrá las cantidades especificadas de rojo, verde y azul. Los métodos **getRed**, **getGreen** y **getBlue** de **Color** devuelven valores enteros entre 0 y 255 que representan la cantidad de rojo, verde y azul, respectivamente. El método **setColor** de **Graphics** establece el color de dibujo actual.

La aplicación de la figura 28.5 muestra varios métodos de la figura 28.4 al dibujar rectángulos llenos y cadenas de diferentes colores.

```

1 // Figura 28.5: MuestraColores.java
2 // Demostración de colores
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6
7 public class MuestraColores extends JFrame {
8     public MuestraColores()
9     {
10         super( "Uso de colores" );
11
12         setSize( 400, 130 );
13         show();
14     } // fin del constructor MuestraColores
15
16     public void paint( Graphics g )
17     {
18         // establece un nuevo color de dibujo por medio de enteros
19         g.setColor( new Color( 255, 0, 0 ) );
20         g.fillRect( 25, 25, 100, 20 );
21         g.drawString( "RVA actual: " + g.getColor(), 130, 40 );
22
23         // establece un nuevo color de dibujo por medio de números de punto
24         // flotante
24         g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
25         g.fillRect( 25, 50, 100, 20 );
26         g.drawString( "RVA actual: " + g.getColor(), 130, 65 );
27
28         // establece un nuevo color de dibujo por medio de objetos estáticos
29         Color
30         g.setColor( Color.blue );
31         g.fillRect( 25, 75, 100, 20 );
32         g.drawString( "RVA actual: " + g.getColor(), 130, 90 );

```

Figura 28.5 Muestra cómo establecer y cómo obtener un **color**. (Parte 1 de 2.)

```

32
33     // despliega valores individuales RGB
34     Color c = Color.magenta;
35     g.setColor( c );
36     g.fillRect( 25, 100, 100, 20 );
37     g.drawString( "valores RVA: " + c.getRed() + ", " +
38                   c.getGreen() + ", " + c.getBlue(), 130, 115 );
39 } // fin del método paint
40
41 public static void main( String args[] )
42 {
43     MuestraColores app = new MuestraColores();
44
45     app.addWindowListener(
46         new WindowAdapter() {
47             public void windowClosing( WindowEvent e )
48             {
49                 System.exit( 0 );
50             } // fin del método windowClosing
51         } // fin de la clase interna anónima
52     ); // fin de addWindowListener
53 } // fin del método main
54 } // fin de la clase MuestraColores

```

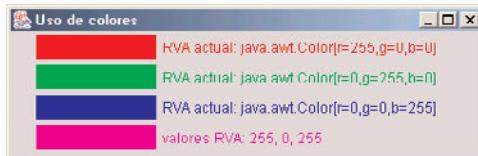


Figura 28.5 Muestra cómo establecer y cómo obtener un **color**. (Parte 2 de 2.)

Cuando comienza la ejecución de la aplicación, se llama al método **paint** de la clase **ShowColors** para pintar la ventana. La línea 19

```
g.setColor( new Color( 255, 0, 0 ) );
```

utiliza el método **setColor** de **Graphics** para establecer el color actual de dibujo. El método **setColor** recibe un objeto **Color**. La expresión **new Color(255, 0, 0)** crea un nuevo objeto **Color** que representa el rojo (valor del rojo **255** y **0** para los colores verde y azul). La línea 20

```
g.fillRect( 25, 25, 100, 20 );
```

utiliza el método **fillRect** de **Graphics** para dibujar un rectángulo relleno con el color actual. Los dos primeros parámetros del método **fillRect** son las coordenadas **x** y **y** de la esquina superior izquierda del rectángulo. El tercer y cuarto parámetros son el ancho y la altura del rectángulo, respectivamente. La línea 21

```
g.drawString( "RVA actual: " + g.getColor(), 130, 40 );
```

utiliza el método **drawString** de **Graphics** para dibujar una cadena (**String**) con el color actual. La expresión **g.getColor()** recibe el color actual desde el objeto **Graphics**. El **Color** devuelto se concatena con la cadena **"RVA actual: "**, lo que resulta en una llamada implícita al método **toString** de la clase **Color**. Observe que la representación de **String** del objeto **Color** contiene el nombre de la clase y el paquete (**java.awt.Color**), y los valores para el rojo, el verde y el azul.

Las líneas 24 a 26 y las líneas 29 a 31 realizan de nuevo las mismas tareas. La línea 24

```
g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
```

utiliza el constructor **Color** con tres argumentos **float** para crear el color verde (**0.0f** para el rojo, **1.0f** para el verde y **0.0f** para el azul). Observe la sintaxis de las constantes. La letra **f** que se agrega a la constan-

te en punto flotante indica que la constante se debe tratar como de tipo **float**. Por lo general, las constantes en punto flotante se tratan como de tipo **double**.

La línea 29 establece el color de dibujo actual en una de las constantes de **Color** predefinidas (**Color.blue**). Observe que el nuevo operador no necesita crear una constante. Las constantes de **Color** son estáticas, de modo que se definen cuando se carga la clase **Color** dentro de memoria en tiempo de ejecución.

La instrucción de las líneas 27 y 38 muestran los métodos **getRed**, **getGreen** y **getBlue** de **Color** y el objeto predefinido **Color.magenta**.

Observación de ingeniería de software 28.2



*Para modificar el color, usted debe crear un objeto **Color** (o utilizar una de las constantes predefinidas de **Color**); no existen métodos **set** (establecer) en la clase **Color** para modificar las características del color actual.*

Una de las más novedosas características de Java es el componente GUI predefinido **JColorChooser** (del paquete **javax.swing**) para la selección de colores. La aplicación de la figura 28.6 le permite oprimir un botón para desplegar un diálogo **JColorChooser**. Cuando selecciona un color y oprime el botón **Aceptar** del diálogo, el color del fondo de la ventana de aplicación cambia.

```

1 // Figura 28.6: MuestraColores2.java
2 // Demostración de JColorChooser
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6
7 public class MuestraColores2 extends JFrame {
8     private JButton cambiaColor;
9     private Color color = Color.lightGray;
10    private Container c;
11
12    public MuestraColores2()
13    {
14        super( "Utilizando JColorChooser" );
15
16        c = getContentPane();
17        c.setLayout( new FlowLayout() );
18
19        cambiaColor = new JButton( "Cambia el color" );
20        cambiaColor.addActionListener(
21            new ActionListener() {
22                public void actionPerformed( ActionEvent e )
23                {
24                    color =
25                        JColorChooser.showDialog( MuestraColores2.this,
26                                      "Elija un color", color );
27
28                    if ( color == null )
29                        color = Color.lightGray;
30
31                    c.setBackground( color );
32                    c.repaint();
33                } // fin del método actionPerformed
34            } // fin de la clase interna anónima
35        ); // fin de addActionListener
36        c.add( cambiaColor );
37
38        setSize( 400, 130 );

```

Figura 28.6 Demostración del diálogo **JColorChooser**. (Parte 1 de 2.)

```

39         show();
40     } // fin del constructor MuestraColores2
41
42     public static void main( String args[ ] )
43     {
44         MuestraColores2 app = new MuestraColores2();
45
46         app.addWindowListener(
47             new WindowAdapter()
48             {
49                 public void windowClosing( WindowEvent e )
50                 {
51                     System.exit( 0 );
52                 } // fin del método windowClosing
53             } // fin de la clase interna anónima
54         ); // fin de addWindowListener
55     } // fin de main
56 } // fin de la clase MuestraColores2

```

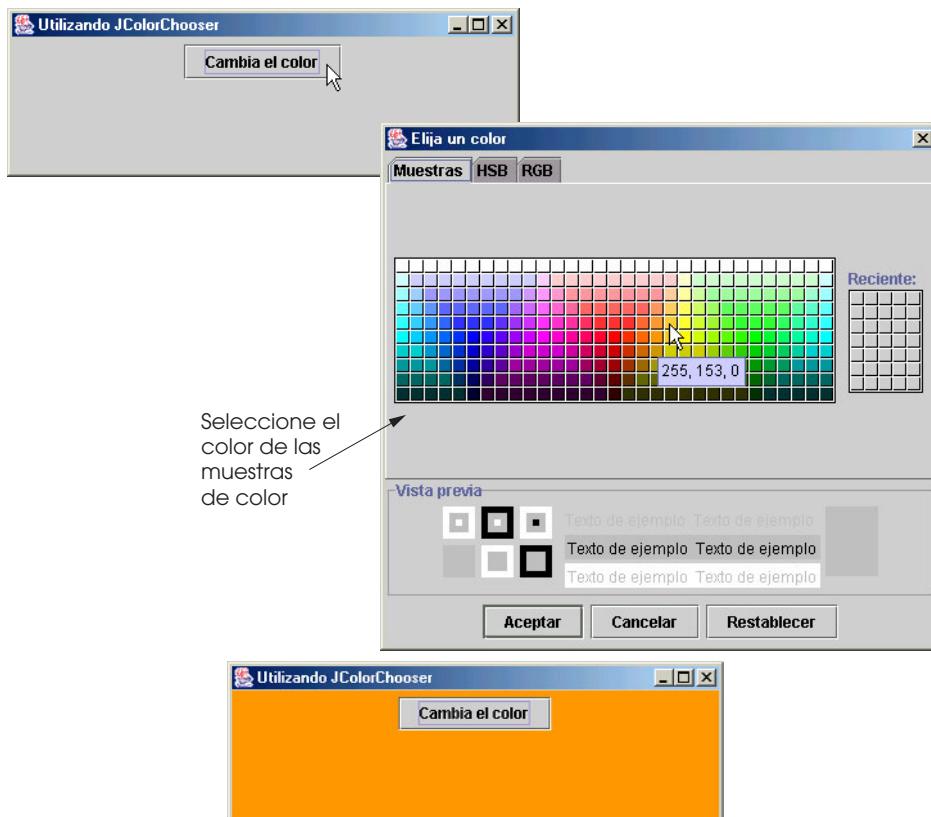


Figura 28.6 Demostración del diálogo **JColorChooser**. (Parte 2 de 2.)

Las líneas 24 a 26 (del método **actionPerformed** para **changeColor**)

```

color =
JColorChooser.showDialog( MuestraColores2.this,
    "Elija un color", color );

```

utilizan el método estático **showDialog** de la clase **JColorChooser** para desplegar el diálogo para la elección de colores. Este método devuelve el **Color** seleccionado (o **null** si el usuario oprime **Cancelar** o cierra el diálogo sin presionar **Aceptar**).

El método **showDialog** toma tres argumentos, una referencia al componente (**Component**) padre, un **String** para desplegar en la barra de título del diálogo y el **Color** seleccionado inicialmente para el diálogo. El componente padre es la ventana desde la cual se despliega el diálogo. Mientras el diálogo de elección de color se encuentre en la pantalla, el usuario no puede interactuar con el componente padre. Este tipo de diálogo se llama *diálogo modal* y lo explicaremos en el capítulo 29. Observe la sintaxis especial **ShowColors2.this** que se utiliza en la instrucción anterior. Cuando utiliza una clase interna, usted puede acceder a la referencia **this** del objeto de la clase externa al calificar a **this** con el nombre de la clase externa y el operador punto (.).

Una vez que el usuario selecciona el color, las líneas 28 y 29 determinan si **color** es **null**, y si es así, establece **color** al **Color.lightGray** predeterminado. La línea 31

```
c.setBackground( color );
```

utiliza el método **setBackground** para modificar el color del fondo del contenido del panel (representado por **Container** **c** en este programa). El método **setBackground** es uno de muchos métodos **Component** que pueden utilizarse en la mayoría de los componentes. La línea 32

```
c.repaint();
```

garantiza que el fondo se repinte al llamar a **repaint** para el panel de contenido. Esto programa una llamada al panel de contenido del método **update** del panel, el cual repinta el fondo del panel de contenido con el color de fondo actual.

La segunda captura de pantalla de la figura 28.6 muestra el diálogo predeterminado **JColorChooser** que permite al usuario seleccionar un color de una variedad de *muestras de colores*. Observe que en realidad existen tres fichas a través de la parte superior del diálogo, **Muestras**, **HSB** y **RGB**. Éstas representan tres diferentes maneras de seleccionar un color. La ficha **HSB** le permite seleccionar un color basado en *tono*, *saturación* y *brillo*. La ficha **RGB** le permite seleccionar un color mediante el uso de barras de desplazamiento para seleccionar los componentes rojo, verde y azul de un color. Las fichas **HSB** y **RGB** aparecen en la figura 28.7.

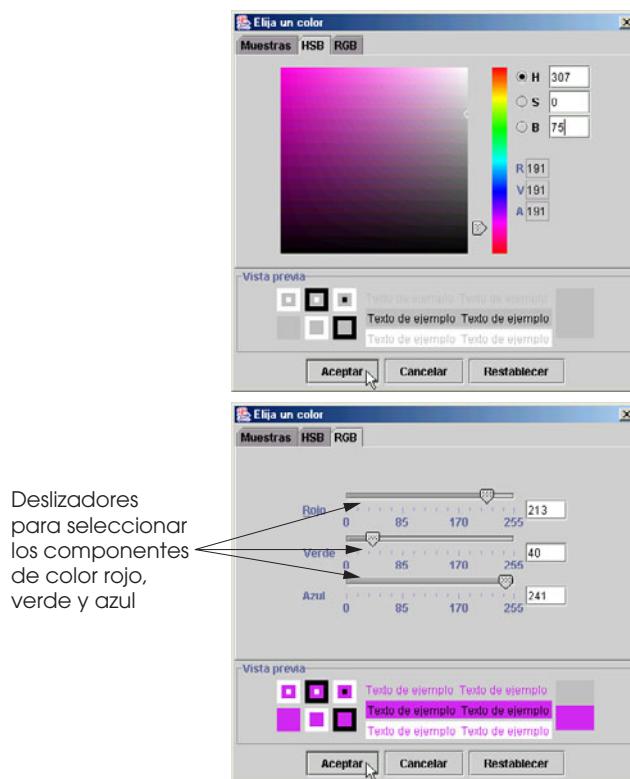


Figura 28.7 Las fichas **HSB** y **RGB** del diálogo **JColorChooser**.

28.4 Control de fuentes

Esta sección presenta los métodos y las constantes para el control de fuentes. La mayoría de los métodos y las constantes de fuentes son parte de la clase **Font**. Algunos métodos de la clase **Font** y de la clase **Graphics** aparecen en la figura 28.8.

El constructor de la clase **Font** toma tres argumentos, el *nombre de la fuente*, el *estilo* y el *tamaño de fuente*. El nombre de la fuente es cualquier fuente soportada por el sistema en donde se ejecuta el programa, tal como las fuentes estándar de Java **Monospaced**, **SansSerif** y **Serif**. El estilo de fuente es **Font.PLAIN**, **Font.ITALIC** o **Font.BOLD** (las constantes estáticas de la clase **Font**). Los estilos de las fuentes pueden utilizarse combinados (por ejemplo, **Font.ITALIC + Font.BOLD**). El tamaño de la fuente se mide en puntos. Un *punto* es 1/72 de una pulgada. El método **setFont** de **Graphics** establece la fuente de dibujo actual en su argumento **Fuente**, es decir, la fuente en la que se desplegará el texto.

Tip de portabilidad 28.2	
	<i>El número de fuentes varía mucho a través de los sistemas. El JDK garantiza que las fuentes Serif, Monospaced, SansSerif, Dialog y DialogInput estarán disponibles.</i>

Método o constante	Descripción
public final static int PLAIN // clase Font	Constante que representa un estilo de fuente común.
public final static int BOLD // clase Font	Constante que representa un estilo de fuente en negritas.
public final static int ITALIC // clase Font	Constante que representa un estilo de fuente en cursivas.
public Font(String nombre, int estilo, int tamaño)	Crea un objeto Font con la fuente, el estilo y el tamaño de la fuente especificada.
public int getStyle() // clase Font	Devuelve un valor entero que indica el estilo de la fuente actual.
public int getSize() // clase Font	Devuelve un valor entero que indica el tamaño de la fuente actual.
public String getName() // clase Font	Devuelve el nombre de la fuente actual como una cadena.
public String getFamily() // clase Font	Devuelve el nombre de la familia de la fuente como una cadena.
public boolean isPlain() // clase Font	Verifica si la fuente es de estilo común. Devuelve true si la fuente es plana.
public boolean isBold() // clase Font	Verifica si la fuente es de estilo negrita. Devuelve true si la fuente es negrita.
public boolean isItalic() // clase Font	Verifica si la fuente es de estilo cursiva. Devuelve true si la fuente es cursiva.
public Font getFont() // clase Graphics	Devuelve un objeto Font que representa la fuente actual.
public void setFont(Font f) // clase Graphics	Establece la fuente actual en la fuente, el estilo y el tamaño determinado por la referencia f al objeto Font .

Figura 28.8 Los métodos de **Font** y las constantes y las fuentes relacionadas con métodos de **Graphics**.

Error común de programación 28.2



Especificar una fuente que no está disponible en un sistema, es un error lógico. Java sustituirá a la fuente predeterminada por el sistema.

El programa de la figura 28.9 despliega texto en cuatro diferentes fuentes con tamaños distintos. El programa utiliza el constructor **Font** para inicializar los objetos **Font** en las líneas 20, 25, 30 y 37 (cada uno en una llamada al método **setFont** de **Graphics** para modificar la fuente a dibujar). Cada llamada al constructor **Font** pasa un nombre de fuente (Serif, Monospaced o SansSerif) como un **String**, un estilo de fuente (**Font.PLAIN**, **Font.ITALIC** o **Font.BOLD**) y el tamaño de la fuente. Una vez que se invoca el método **setFont** de **Graphics**, todo el texto que se despliegue después de la llamada aparecerá con la nueva fuente hasta que ésta se modifique. Observe que la línea 35 modifica el color del dibujo a rojo, de modo que la siguiente cadena que se despliega aparece en rojo.

Observación de ingeniería de software 28.3



*Para modificar la fuente, debe crear un nuevo objeto **Font**; no existen métodos establecer (set) en la clase **Font** para modificar las características de la fuente actual.*

```

1 // Figura 28.9: Fuentes.java
2 // Uso de fuentes
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6
7 public class Fuentes extends JFrame {
8     public Fuentes()
9     {
10         super( "Utilizando fuentes" );
11
12         setSize( 420, 125 );
13         show();
14     } // fin del constructor Fuentes
15
16     public void paint( Graphics g )
17     {
18         // establece la fuente actual en Serif (Times), negrita, 12pt
19         // y dibuja una cadena
20         g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
21         g.drawString( "Serif de 12 puntos en negritas.", 20, 50 );
22
23         // establece la fuente actual en Monospaced (Courier),
24         // cursiva, 24pt and draw a string
25         g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
26         g.drawString( "Monospaced de 24 puntos en cursivas.", 20, 70 );
27
28         // establece la fuente actual en SansSerif (Helvetica),
29         // en texto común, 14pt y dibuja una cadena
30         g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
31         g.drawString( "SansSerif de 14 puntos en texto comun.", 20, 90 );
32
33         // establece la fuente actual en Serif (times), negritas/cursivas,
34         // de 18pt y dibuja una cadena
35         g.setColor( Color.red );
36         g.setFont(
37             new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
38         g.drawString( g.getFont().getName() + " " +

```

Figura 28.9 Uso del método **setFont** de **Graphics** para modificar las **Fuentes**. (Parte 1 de 2.)

```

39                     g.getFont().getSize() +
40                         " puntos en negritas y cursivas.", 20, 110 );
41     } // fin del método paint
42
43     public static void main( String args[] )
44     {
45         Fuentes app = new Fuentes();
46
47         app.addWindowListener(
48             new WindowAdapter() {
49                 public void windowClosing( WindowEvent e )
50                 {
51                     System.exit( 0 );
52                 } // fin del método windowClosing
53             } // fin de la clase interna anónima
54         ); // fin de addWindowListener
55     } // fin de main
56 } // fin de la clase Fuentes

```

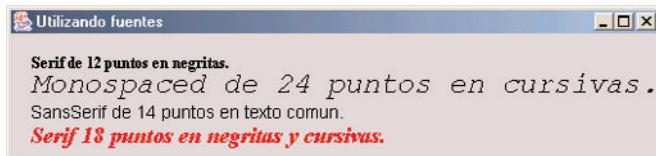


Figura 28.9 Uso del método **setFont** de **Graphics** para modificar las **Fuentes**. (Parte 2 de 2.)

Con frecuencia es necesario obtener información acerca de la fuente actual, tal como el nombre, el estilo y el tamaño de la fuente. Muchos métodos de **Font** utilizados para obtener información de la fuente aparecen en la figura 28.8. El método **getStyle** devuelve un valor entero que representa el estilo actual. El valor entero devuelto es **Font.PLAIN**, **Font.ITALIC**, **Font.BOLD** o cualquier combinación de **Font.PLAIN**, **Font.ITALIC** y **Font.BOLD**.

El método **getSize** devuelve el tamaño de la fuente en puntos. El método **getName** devuelve el nombre de la fuente actual como un **String**. El método **getFamily** devuelve el nombre de la familia de la fuente a la que pertenece la fuente. El nombre de la familia de la fuente es específica de la plataforma.

Tip de portabilidad 28.3



Java utiliza nombres de fuentes estandarizados y los mapea en sistemas específicos de nombres de fuentes para portabilidad. Esto es transparente para el programador.

Los métodos de **Font** también están disponibles para evaluar el estilo de la fuente actual y se resumen en la figura 28.8. El método **isPlain** devuelve **true** si el estilo de fuente actual es común (plano). El método **isBold** devuelve **true** si el estilo de la fuente actual es en negritas. El método **isItalic** devuelve **true** si el estilo de la fuente actual es en cursivas.

En ocasiones, es necesario conocer la información precisa acerca de la métrica de una fuente, tal como la *altura*, el *descendente* (la cantidad de puntos de carácter por debajo de la línea base), el *ascendente* (la cantidad de puntos de carácter por arriba de la línea base) y el *interlineado* (la diferencia entre la altura y el ascendente). La figura 28.10 muestra algunas métricas comunes de las fuentes. Observe que la coordenada pasada a **drawString** corresponde a la esquina inferior izquierda de la línea base de la fuente.

La clase **FontMetrics** define diversos métodos para obtener las características de una fuente. Estos métodos, así como el método **getFontMetrics** de **Graphics**, se encuentran resumidos en la figura 28.11.

El programa de la figura 28.12 utiliza los métodos de la figura 28.11 para obtener información sobre la métrica de dos fuentes.

La línea 19 crea y establece la fuente de dibujo actual en **SansSerif** de 12 puntos en negritas. La línea 20 utiliza el método **getFontMetrics** de **Graphics** para obtener el objeto **FontMetrics** para la fuente

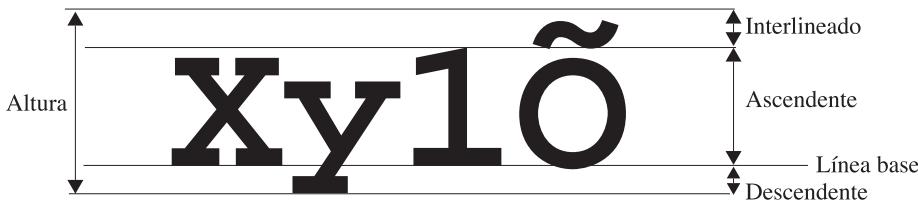


Figura 28.10 Métrica de una fuente.

Método	Descripción
<code>public int getAscent()</code>	//clase FontMetrics Devuelve un valor que representa el ascendente en puntos de una fuente.
<code>public int getDescent()</code>	//clase FontMetrics Devuelve un valor que representa el descendente en puntos de una fuente.
<code>public int getLeading()</code>	//clase FontMetrics Devuelve un valor que representa el interlineado en puntos de una fuente.
<code>public int getHeight()</code>	//clase FontMetrics Devuelve un valor que representa la altura en puntos de una fuente.
<code>public FontMetrics getFontMetrics()</code>	//clase Graphics Devuelve un valor que representa la altura en puntos de una fuente.
<code>public FontMetrics getFontMetrics(Font f)</code>	//clase Graphics Devuelve el objeto FontMetrics para el argumento especificado Font .

Figura 28.11 Métodos **FontMetrics** y **Graphics** para obtener la métrica de una fuente.

```

1 // Figura 28.12: Metrica.java
2 // Demostración de los métodos de la clase FontMetrics y
3 // de la clase Graphics que son útiles para obtener la métrica de una fuente
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.swing.*;
7
8 public class Metrica extends JFrame {
9     public Metrica()
10    {
11        super( "Demostrando FontMetrics" );
12
13        setSize( 510, 210 );
14        show();
15    } // fin del constructor Metrica
16
17    public void paint( Graphics g )
18    {
19        g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
20        FontMetrics fm = g.getFontMetrics();
21        g.drawString( "Fuente actual: " + g.getFont(), 10, 40 );
22        g.drawString( "Ascendente: " + fm.getAscent(), 10, 55 );
23        g.drawString( "Descendente: " + fm.getDescent(), 10, 70 );

```

Figura 28.12 Cómo obtener información sobre la métrica de una fuente. (Parte 1 de 2.)

```

24     g.drawString( "Altura: " + fm.getHeight(), 10, 85 );
25     g.drawString( "Interlineado: " + fm.getLeading(), 10, 100 );
26
27     Font fuente = new Font( "Serif", Font.ITALIC, 14 );
28     fm = g.getFontMetrics( fuente );
29     g.setFont( fuente );
30     g.drawString( "Fuente actual: " + fuente, 10, 130 );
31     g.drawString( "Ascendente: " + fm.getAscent(), 10, 145 );
32     g.drawString( "Descendente: " + fm.getDescent(), 10, 160 );
33     g.drawString( "Altura: " + fm.getHeight(), 10, 175 );
34     g.drawString( "Interlineado: " + fm.getLeading(), 10, 190 );
35 } // fin del método paint
36
37 public static void main( String args[] )
38 {
39     Metrica app = new Metrica();
40
41     app.addWindowListener(
42         new WindowAdapter() {
43             public void windowClosing( WindowEvent e )
44             {
45                 System.exit( 0 );
46             } // fin del método windowClosing
47         } // fin de la clase interna anónima
48     ); // fin de addWindowListener
49 } // fin de main
50 } // fin de la clase Metrica

```

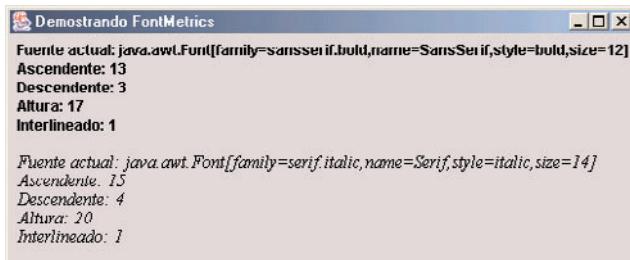


Figura 28.12 Cómo obtener información sobre la métrica de una fuente. (Parte 2 de 2.)

actual. La línea 21 utiliza una llamada implícita al método **toString** de la clase **Font** para desplegar la representación de la cadena de la fuente. Las líneas 22 a 25 utilizan los métodos **FontMetric** para obtener el ascendente, el descendente, la altura y el interlineado de la fuente.

La línea 27 crea una nueva fuente **Serif** de 14 puntos en cursivas. La línea 28 utiliza una segunda versión del método **getFontMetrics** de **Graphics**, el cual recibe un argumento **Font** y devuelve un objeto **FontMetrics** correspondiente. Las líneas 31 a 34 obtienen el ascendente, el descendente, la altura y el interlineado para la fuente. Observe que las métricas de la fuente son ligeramente distintas para las dos fuentes.

28.5 Cómo dibujar líneas, rectángulos y elipses

Esta sección presenta una variedad de métodos **Graphics** para dibujar líneas, rectángulos y elipses. Los métodos y sus parámetros aparecen resumidos en la figura 28.13. Para cada método de dibujo que requiera un parámetro **ancho** y **altura**, estos valores deben ser positivos. De lo contrario, la figura no se desplegará.

La aplicación de la figura 28.14 muestra el dibujo de una variedad de líneas, rectángulos, rectángulos tridimensionales, rectángulos redondeados y elipses.

Método	Descripción
<code>public void drawLine(int x1, int y1, int x2, int y2)</code>	Dibuja una línea entre el punto (<code>x1, y1</code>) y el punto (<code>x2, y2</code>).
<code>public void drawRect(int x, int y, int ancho, int altura)</code>	Dibuja un rectángulo con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas (<code>x, y</code>).
<code>public void fillRect(int x, int y, int ancho, int altura)</code>	Dibuja un rectángulo sólido con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas (<code>x, y</code>).
<code>public void clearRect(int x, int y, int ancho, int altura)</code>	Dibuja un rectángulo sólido con el ancho y la altura especificados en el color de fondo actual. La esquina superior izquierda del rectángulo tiene las coordenadas (<code>x, y</code>).
<code>public void drawRoundRect(int x, int y, int ancho, int altura, int anchoArco, int alturaArco)</code>	Dibuja un rectángulo con las esquinas redondeadas en el color actual con el ancho y la altura especificados. Los argumentos <code>anchoArco</code> y <code>alturaArco</code> determinan el redondeo de las esquinas (vea la figura 28.15).
<code>public void fillRoundRect(int x, int y, int ancho, int altura, int anchoArco, int alturaArco)</code>	Dibuja un rectángulo sólido con las esquinas redondeadas en el color actual con el ancho y la altura especificados. Los argumentos <code>anchoArco</code> y <code>alturaArco</code> determinan el redondeo de las esquinas (vea la figura 28.15).
<code>public void draw3DRect(int x, int y, int ancho, int altura, Boolean b)</code>	Dibuja un rectángulo tridimensional en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas (<code>x, y</code>). El rectángulo aparece aumentado cuando <code>b</code> es <code>true</code> y disminuido cuando <code>b</code> es <code>false</code> .
<code>public void fill3DRect(int x, int y, int ancho, int altura, Boolean b)</code>	Dibuja un rectángulo relleno tridimensional en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo tiene las coordenadas (<code>x, y</code>). El rectángulo aparece aumentado cuando <code>b</code> es <code>true</code> y disminuido cuando <code>b</code> es <code>false</code> .
<code>public void drawOval(int x, int y, int ancho, int altura)</code>	Dibuja una elipse en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo delimitador se encuentra en las coordenadas (<code>x, y</code>). La elipse toca los cuatro lados del rectángulo delimitador, en el centro de cada lado (vea la figura 28.16).
<code>public void fillOval(int x, int y, int ancho, int altura)</code>	Dibuja una elipse rellena en el color actual con el ancho y la altura especificados. La esquina superior izquierda del rectángulo delimitador se encuentra en las coordenadas (<code>x, y</code>). La elipse toca los cuatro lados del rectángulo delimitador, en el centro de cada lado (vea la figura 28.16).

Figura 28.13 Métodos `Graphics` que dibujan líneas, rectángulos y elipses.

```

1 // Figura 28.14: LineasRectangsElips.java
2 // Dibuja líneas, rectángulos y elipses
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;

```

Figura 28.14 Demostración del método `drawLine` de `Graphics`. (Parte 1 de 2.)

```

6
7  public class LineasRectangsElips extends JFrame {
8      private String s = "Utilizando drawString!";
9
10     public LineasRectangsElips()
11     {
12         super( "Dibujando lineas, rectangulos y elipses" );
13
14         setSize( 400, 165 );
15         show();
16     } // fin del constructor LineasRectangsElips
17
18     public void paint( Graphics g )
19     {
20         g.setColor( Color.red );
21         g.drawLine( 5, 30, 350, 30 );
22
23         g.setColor( Color.blue );
24         g.drawRect( 5, 40, 90, 55 );
25         g.fillRect( 100, 40, 90, 55 );
26
27         g.setColor( Color.cyan );
28         g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
29         g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
30
31         g.setColor( Color.yellow );
32         g.draw3DRect( 5, 100, 90, 55, true );
33         g.fill3DRect( 100, 100, 90, 55, false );
34
35         g.setColor( Color.magenta );
36         g.drawOval( 195, 100, 90, 55 );
37         g.fillOval( 290, 100, 90, 55 );
38     } // fin del método paint
39
40     public static void main( String args[] )
41     {
42         LineasRectangsElips app = new LineasRectangsElips();
43
44         app.addWindowListener(
45             new WindowAdapter() {
46                 public void windowClosing( WindowEvent e )
47                 {
48                     System.exit( 0 );
49                 } // fin del método windowClosing
50             } // fin de la clase interna anónima
51         ); // fin de addWindowListener
52     } // fin de main
53 } // fin de la clase LineasRectangsElips

```

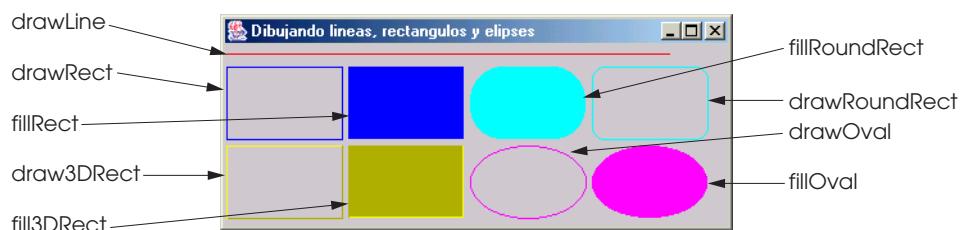


Figura 28.14 Demostración del método **drawLine** de **Graphics**. (Parte 2 de 2.)

Los métodos **fillRoundRect** (línea 28) y **drawRoundRect** (línea 29) dibujan rectángulos con esquinas redondeadas. Sus dos primeros argumentos especifican las coordenadas de la esquina superior izquierda del rectángulo delimitador, es decir, el área en la que se dibujará el rectángulo. Observe que las coordenadas de la esquina superior izquierda no corresponden al borde del rectángulo redondeado, sino a las coordenadas en donde estaría el borde si el rectángulo tuviera esquinas cuadradas. El tercero y cuarto argumentos especifican el **ancho** y la **altura** del rectángulo. Sus dos últimos argumentos, **anchoArco** y **alturaArco**, determinan los diámetros horizontal y vertical de los arcos utilizados para representar las esquinas.

Los métodos **draw3DRect** (línea 32) y **fill3DRect** (línea 33) toman los mismos argumentos. Los dos primeros argumentos especifican la esquina superior izquierda del rectángulo. Los dos siguientes argumentos especifican el **ancho** y la **altura** del rectángulo, respectivamente. El último argumento determina si el rectángulo es *aumentado* (**true**) o *disminuido* (**false**). El efecto tridimensional de **draw3DRect** aparece como dos bordes del rectángulo en el color original y dos bordes en un color ligeramente más oscuro. El efecto tridimensional de **fill3DRect** aparece como dos bordes del rectángulo en el color original de dibujo, y el relleno y los otros dos bordes en un color ligeramente más oscuro. Los rectángulos aumentados tienen el borde superior y el de la izquierda con el color original de dibujo. Los rectángulos disminuidos tienen el borde inferior y el de la derecha con el color original de dibujo. El efecto tridimensional es difícil de apreciar en algunos colores.

La figura 28.15 etiqueta el ancho del arco, la altura del arco, el ancho y la altura de un rectángulo redondeado. Si se utiliza el mismo valor para **anchoArco** y **alturaArco**, se produce un cuarto de círculo en cada esquina. Cuando **ancho**, **altura**, **anchoArco** y **alturaArco** tienen los mismos valores, el resultado es un círculo. Si los valores de **ancho** y **altura** son los mismos, y los valores de **anchoArco** y **alturaArco** son 0, el resultado es un cuadrado.

Tanto el método **drawOval** como **fillOval** toman los mismos cuatro argumentos. Los dos primeros argumentos especifican la coordenada superior izquierda del rectángulo delimitador que contiene la elipse. Los dos últimos argumentos especifican el ancho y la altura del rectángulo delimitador, respectivamente. La figura 28.16 muestra una elipse (óvalo) delimitado por un rectángulo. Observe que la elipse toca el centro de los cuatro lados del rectángulo delimitador (el rectángulo delimitador no se despliega en la pantalla).

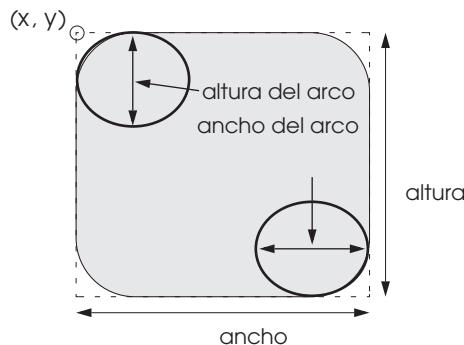


Figura 28.15 El ancho y la altura del arco para rectángulos redondeados.

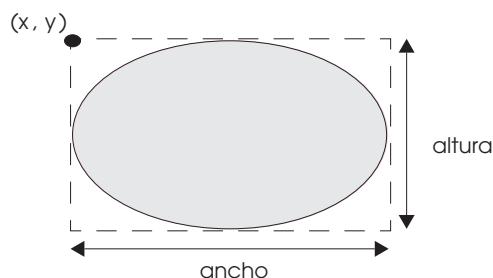


Figura 28.16 Una elipse limitada por un rectángulo.

28.6 Cómo dibujar arcos

Un *arco* es una parte de una elipse. Los ángulos de un arco se miden en grados. Los arcos *barren* desde un *ángulo inicial* el número de grados especificados por al *ángulo del arco*. El ángulo de inicio indica en grados en dónde comienza el arco. El ángulo del arco especifica el número total de grados que el arco barre. La figura 28.17 muestra dos arcos. El conjunto izquierdo de ejes muestra un arco que barre desde cero hasta aproximadamente 110 grados. Los arcos que barren en contra de las manecillas del reloj se miden con *grados positivos*. El conjunto derecho de ejes muestran un arco que barre desde cero hasta aproximadamente 110 grados. Los arcos que barren en la dirección de las manecillas del reloj se miden con *grados negativos*. Observe los cuadros punteados alrededor de los arcos de la figura 28.17. Cuando dibujamos un arco, especificamos un rectángulo delimitador para una elipse. El arco barrerá parte de la elipse. Los métodos **drawArc** y **fillArc** de **Graphics** para dibujar los arcos aparecen en la figura 28.18.



Figura 28.17 Arcos con ángulos positivos y negativos.

Método	Descripción
<code>public void drawArc(int x, int y, int ancho, int altura, int anguloInicial, int anguloArco)</code>	Dibuja un arco relativo a la esquina superior izquierda (x , y) del rectángulo inscrito con el ancho y la altura especificados. El segmento de arco se dibuja a partir de anguloInicial y barre el número de grados indicado por anguloArco .
<code>public void fillArc(int x, int y, int ancho, int altura, int anguloInicial, int anguloArco)</code>	Dibuja un arco sólido (es decir, un sector) relativo a la esquina superior izquierda (x , y) del rectángulo inscrito con el ancho y la altura especificados. El segmento de arco se dibuja comenzando en anguloInicial y barre el número de grados indicado por anguloArco .

Figura 28.18 Métodos de **Graphics** para dibujar arcos.

El programa de la figura 28.19 muestra los métodos para arcos de la figura 28.18. El programa dibuja seis arcos (tres arcos vacíos y tres arcos rellenos). Para mostrar el rectángulo delimitador que determina en dónde aparece el arco, los tres primeros arcos se despliegan dentro de un rectángulo amarillo que tiene los mismos argumentos **x**, **y**, **ancho** y **altura** como arcos.

```

1 // Figura 28.19: DibujoArcos.java
2 // Cómo dibujar arcos
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;

```

Figura 28.19 Demostración de **drawArc** y **fillArc**. (Parte 1 de 3.)

```
6
7  public class DibujoArcos extends JFrame {
8      public DibujoArcos()
9      {
10         super( "Dibujando arcos" );
11
12         setSize( 300, 170 );
13         show();
14     } // fin del constructor DibujoArcos
15
16     public void paint( Graphics g )
17     {
18         // comienza en 0 y barre 360 grados
19         g.setColor( Color.yellow );
20         g.drawRect( 15, 35, 80, 80 );
21         g.setColor( Color.black );
22         g.drawArc( 15, 35, 80, 80, 0, 360 );
23
24         // comienza en 0 y barre 110 grados
25         g.setColor( Color.yellow );
26         g.drawRect( 100, 35, 80, 80 );
27         g.setColor( Color.black );
28         g.drawArc( 100, 35, 80, 80, 0, 110 );
29
30         // comienza en 0 y barre -270 grados
31         g.setColor( Color.yellow );
32         g.drawRect( 185, 35, 80, 80 );
33         g.setColor( Color.black );
34         g.drawArc( 185, 35, 80, 80, 0, -270 );
35
36         // comienza en 0 y barre 360 grados
37         g.fillArc( 15, 120, 80, 40, 0, 360 );
38
39         // comienza en 270 y barre -90 grados
40         g.fillArc( 100, 120, 80, 40, 270, -90 );
41
42         // comienza en 0 y barre -270 grados
43         g.fillArc( 185, 120, 80, 40, 0, -270 );
44     } // fin del método paint
45
46     public static void main( String args[] )
47     {
48         DibujoArcos app = new DibujoArcos();
49
50         app.addWindowListener(
51             new WindowAdapter() {
52                 public void windowClosing( WindowEvent e )
53                 {
54                     System.exit( 0 );
55                 } // fin del método windowClosing
56             } // fin de la clase interna anónima
57         ); // fin de addWindowListener
58     } // fin de main
59 } // fin de la clase DibujoArcos
```

Figura 28.19 Demostración de **drawArc** y **fillArc**. (Parte 2 de 3.)

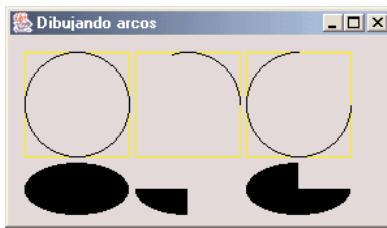


Figura 28.19 Demostración de `drawArc` y `fillArc`. (Parte 3 de 3.)

28.7 Cómo dibujar polígonos y polilíneas

Los *polígonos* son figuras con múltiples lados. Las *polilíneas* son una serie de puntos conectados. Los métodos gráficos para dibujar polígonos y polilíneas aparecen en la figura 28.19. Observe que algunos métodos requieren un objeto `Polygon` (del paquete `java.awt`). Los constructores de la clase `Polygon` también aparecen en la figura 28.20.

Método	Descripción
<code>public void drawPolygon (int puntosX[], int puntosY[], int puntos)</code>	Dibuja un polígono. La coordenada <i>x</i> para cada punto se especifica en el arreglo <code>puntosX</code> y la coordenada <i>y</i> de cada punto se especifica en el arreglo <code>puntosY</code> . El último argumento especifica el número de <code>puntos</code> . Este método dibuja un polígono cerrado, incluso si el último punto es diferente del primer punto.
<code>public void drawPolyline (int puntosX[], int puntosY[], int puntos)</code>	Dibuja una serie de líneas conectadas. La coordenada <i>x</i> de cada punto se especifica en el arreglo <code>puntosX</code> y la coordenada <i>y</i> de cada punto se especifica en el arreglo <code>puntosY</code> . El último argumento especifica el número de <code>puntos</code> . Si el último punto es diferente del primer punto, la polilínea no se cierra.
<code>public void drawPolygon (Polygon g)</code>	Dibuja el polígono cerrado especificado.
<code>public void fillPolygon (int puntosX[], int puntosY[], int puntos)</code>	Dibuja un polígono sólido. La coordenada <i>x</i> de cada punto se especifica en el arreglo <code>puntosX</code> y la coordenada <i>y</i> de cada punto se especifica en el arreglo <code>puntosY</code> . El último argumento especifica el número de <code>puntos</code> . Este método dibuja un polígono cerrado, incluso si el último punto es diferente del primer punto.
<code>public void fillPolygon(Polygon p)</code>	Dibuja el polígono sólido especificado. El polígono es cerrado.
<code>public Polygon()</code>	Construye un nuevo objeto polígono. El polígono no contiene punto alguno.
<code>public Polygon (int valoresX[], int valoresY[], int numeroDePuntos) // clase Polygon</code>	Construye un nuevo objeto polígono. El polígono tiene el número de lados que especifica <code>numeroDePuntos</code> , en donde cada punto consta de una coordenada <i>x</i> correspondiente a <code>valoresX</code> , y una coordenada <i>y</i> correspondiente a <code>valoresY</code> .

Figura 28.20 Los métodos de `Graphics` para dibujar polígonos, y los constructores de la clase `Polygon`

El programa de la figura 28.21 dibuja polígonos y polilíneas por medio de los métodos y constructores de la figura 28.20.

```
1 // Figura 28.21: DibujoPoligonos.java
2 // Cómo dibujar polígonos
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class DibujoPoligonos extends JFrame {
8     public DibujoPoligonos()
9     {
10         super( "Dibujando polígonos" );
11
12         setSize( 275, 230 );
13         show();
14     } // fin del constructor DibujoPoligonos
15
16     public void paint( Graphics g )
17     {
18         int valoresX[] = { 20, 40, 50, 30, 20, 15 };
19         int valoresY[] = { 50, 50, 60, 80, 80, 60 };
20         Polygon poli1 = new Polygon( valoresX, valoresY, 6 );
21
22         g.drawPolygon( poli1 );
23
24         int valoresX2[] = { 70, 90, 100, 80, 70, 65, 60 };
25         int valoresY2[] = { 100, 100, 110, 110, 130, 110, 90 };
26
27         g.drawPolyline( valoresX2, valoresY2, 7 );
28
29         int valoresX3[] = { 120, 140, 150, 190 };
30         int valoresY3[] = { 40, 70, 80, 60 };
31
32         g.fillPolygon( valoresX3, valoresY3, 4 );
33
34         Polygon poli2 = new Polygon();
35         poli2.addPoint( 165, 135 );
36         poli2.addPoint( 175, 150 );
37         poli2.addPoint( 270, 200 );
38         poli2.addPoint( 200, 220 );
39         poli2.addPoint( 130, 180 );
40
41         g.fillPolygon( poli2 );
42     } // fin del método paint
43
44     public static void main( String args[] )
45     {
46         DibujoPoligonos app = new DibujoPoligonos();
47
48         app.addWindowListener(
49             new WindowAdapter() {
50                 public void windowClosing( WindowEvent e )
51                 {
52                     System.exit( 0 );
53                 } // fin del método windowClosing
54             }
55         );
56     }
57 }
```

Figura 28.21 Demostración de **drawPolygon** y **fillPolygon**. (Parte 1 de 2.)

```

54 } // fin de la clase interna anónima
55 ); // fin de addWindowListener
56 } // fin de main
57 } // fin de la clase DibujoPoligonos

```

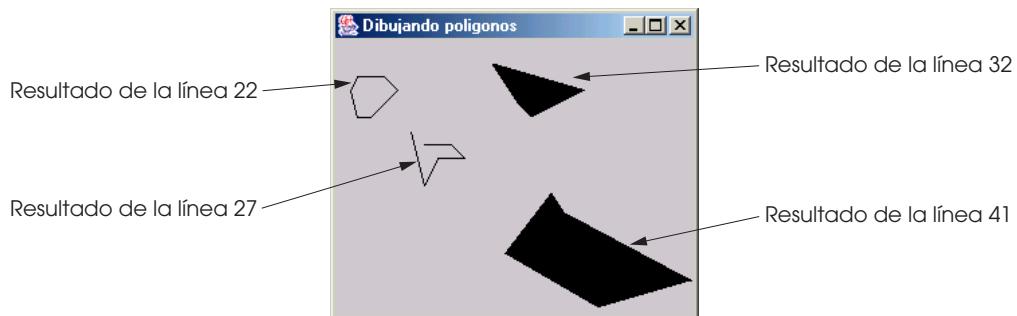


Figura 28.21 Demostración de `drawPolygon` y `fillPolygon`. (Parte 2 de 2.)

Las líneas 18 a 20 crean dos arreglos `int` y los utilizan para especificar los puntos para `Polygon poli1`. La llamada al constructor de `Polygon` en la línea 20 recibe el arreglo `valoresX`, el cual contiene la coordenada *x* de cada punto, el arreglo `valoresY`, el cual contiene la coordenada *y* de cada punto, y 6 (el número de puntos en el polígono). La línea 22 despliega `poli1`, pasándolo como un argumento del método `drawPolygon` de `Graphics`.

Las líneas 24 y 25 crean dos arreglos enteros, y los utilizan para especificar los puntos de una serie de líneas conectadas. El arreglo `valoresX2` contiene la coordenada *x* de cada punto, y el arreglo `valoresY2` contiene la coordenada *y* de cada punto. La línea 27 utiliza el método `drawPolyline` de `Graphics` para desplegar la serie de líneas conectadas, especificadas con los argumentos `valoresX2`, `valoresY2` y 7 (el número de puntos).

Las líneas 29 y 30 crean dos arreglos enteros, y los utilizan para especificar los puntos de un polígono. El arreglo `valoresX3` contiene la coordenada *x* de cada punto, y el arreglo `valoresY3` contiene la coordenada *y* de cada punto. La línea 32 despliega un polígono, pasando al método `fillPolygon` de `Graphics` los dos arreglos (`valoresX3` y `valoresY3`) y el número de puntos a dibujar (4).



Error común de programación 28.3

Si el número de puntos especificados en el tercer argumento del método `drawPolygon` o del método `fillPolygon` es mayor que el número de elementos de los arreglos de coordenadas que definen el polígono a desplegar, se lanza una `ArrayIndexOutOfBoundsException`.

La línea 34 crea `poli2` de `Polygon` sin puntos. Las líneas 35 a 39 utilizan el método `addPoint` de `Polygon` para agregar pares de coordenadas *x* y *y* al polígono. La línea 41 despliega `poli2` de `Polygon`, pasándolo al método `fillPolygon` de `Graphics`.

28.8 La API Java2D

El API Java2D proporciona capacidades para gráficos de dos dimensiones a programadores que requieren manipulaciones gráficas detalladas y complejas. La API incluye características para el procesamiento de líneas de arte, texto e imágenes de los paquetes `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` y `java.awt.image.renderable`. Las capacidades de la API son demasiado extensas como para cubrirlas en este texto. En esta sección, presentamos una perspectiva general de diversas capacidades de Java2D.

Dibujar con la API Java2D se logra con una instancia de la clase `Graphics2D` (del paquete `java.awt`). La clase `Graphics2D` es una subclase de la clase `Graphics`, por lo que tiene todas las capacidades gráficas que mostramos anteriormente en este capítulo. De hecho, el objeto real que utilizamos para dibujar en cada método `paint` es `Graphics2D` que se pasa al método `paint`, y se accede a él a través de la referencia de

superclase **g** de **Graphics**. Para acceder a las capacidades de **Graphics2D**, debemos convertir el tipo de la referencia de **Graphics** pasada a **paint** en una referencia **Graphics2D** con una instrucción como

```
Graphics2D g2d = ( Graphics2D ) g;
```

Los programas de las siguientes secciones utilizan esta técnica.

28.9 Figuras en Java2D

A continuación, presentamos diversas figuras Java2D del paquete **java.awt.geom**, que incluyen **Ellipse2D.Double**, **Rectangle2D.Double**, **Arc2D.Double**, **Line2D.Double** y **RoundRectangle2D.Double**. Observe la sintaxis del nombre de cada clase. Cada una de estas clases representa una figura con dimensiones especificadas como valores de punto flotante de precisión doble. Existe una versión aparte de cada una, representada con valores de punto flotante de precisión simple (como **Ellipse2D.Float**). En cada caso, **Double** es una clase interna estática de la clase que se encuentra a la izquierda del operador punto (por ejemplo, **Ellipse2D**). Para utilizar la clase interna estática, simplemente calificamos su nombre con el nombre de la clase externa.

El programa de la figura 28.22 demuestra diversas figuras Java2D y dibuja características como líneas gruesas, rellena figuras con patrones, y dibuja líneas punteadas. Éstas son sólo algunas de las diversas capacidades provistas por Java2D.

```

1 // Figura 28.22: Figuras.java
2 // Demostración de algunas figuras de Java2D
3 import javax.swing.*;
4 import java.awt.event.*;
5 import java.awt.*;
6 import java.awt.geom.*;
7 import java.awt.image.*;
8
9 public class Figuras extends JFrame {
10     public Figuras()
11     {
12         super( "Dibujando figuras de 2D" );
13
14         setSize( 425, 160 );
15         show();
16     } // fin del constructor Figuras
17
18     public void paint( Graphics g )
19     {
20         // crea una figura en 2D convirtiendo el tipo de g a Graphics2D
21         Graphics2D g2d = ( Graphics2D ) g;
22
23         // dibuja una elipse en 2D rellena con un degradado azul-amarillo
24         g2d.setPaint(
25             new GradientPaint( 5, 30,           // x1, y1
26                               Color.blue,    // Color inicial
27                               35, 100,       // x2, y2
28                               Color.yellow, // fin de Color
29                               true );        // cíclico
30         g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
31
32         // dibuja un rectángulo en 2D en color rojo
33         g2d.setPaint( Color.red );

```

Figura 28.22 Demostración de algunas figuras de Java2D. (Parte 1 de 3.)

```
34     g2d.setStroke( new BasicStroke( 10.0f ) );
35     g2d.draw(
36         new Rectangle2D.Double( 80, 30, 65, 100 ) );
37
38     // dibuja un rectángulo redondeado en 2D con un fondo con
39     // buferbuffered background
40     BufferedImage imagenBuf =
41         new BufferedImage(
42             10, 10, BufferedImage.TYPE_INT_RGB );
43
44     Graphics2D gg = imagenBuf.createGraphics();
45     gg.setColor( Color.yellow ); // dibuja en amarillo
46     gg.fillRect( 0, 0, 10, 10 ); // dibuja un rectángulo relleno
47     gg.setColor( Color.black ); // dibuja en negro
48     gg.drawRect( 1, 1, 6, 6 ); // dibuja un rectángulo
49     gg.setColor( Color.blue ); // dibuja en azul
50     gg.fillRect( 1, 1, 3, 3 ); // dibuja un rectángulo relleno
51     gg.setColor( Color.red ); // dibuja en rojo
52     gg.fillRect( 4, 4, 3, 3 ); // dibuja un rectángulo relleno
53
54     // pinta la imagenBuf en el JFrame
55     g2d.setPaint(
56         new TexturePaint(
57             imagenBuf, new Rectangle( 10, 10 ) ) );
58     g2d.fill(
59         new RoundRectangle2D.Double(
60             155, 30, 75, 100, 50, 50 ) );
61
62     // dibuja un arco en 2D en forma de pastel en color blanco
63     g2d.setPaint( Color.white );
64     g2d.setStroke( new BasicStroke( 6.0f ) );
65     g2d.draw(
66         new Arc2D.Double(
67             240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
68
69     // dibuja líneas en 2D en verde y amarillo
70     g2d.setPaint( Color.green );
71     g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
72
73     float guiones[] = { 10 };
74
75     g2d.setPaint( Color.yellow );
76     g2d.setStroke(
77         new BasicStroke( 4,
78             BasicStroke.CAP_ROUND,
79             BasicStroke.JOIN_ROUND,
80             10, guiones, 0 ) );
81     g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
82 } // fin del método paint
83
84 public static void main( String args[] )
85 {
86     Figuras app = new Figuras();
```

Figura 28.22 Demostración de algunas figuras de Java2D. (Parte 2 de 3.)

```

87     app.addWindowListener(
88         new WindowAdapter() {
89             public void windowClosing( WindowEvent e ) {
90                 {
91                     System.exit( 0 );
92                 } // fin del método windowClosing
93             } // fin de la clase interna anónima
94         ); // fin de addWindowListener
95     } // fin de main
96 } // fin de la clase Figuras

```

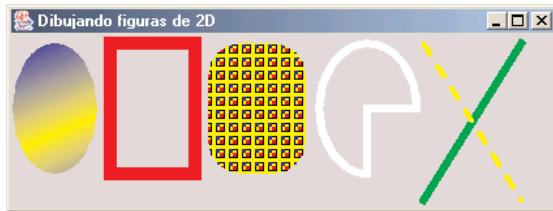


Figura 28.22 Demostración de algunas figuras de Java2D. (Parte 3 de 3.)

La línea 21 convierte el tipo de la referencia **Graphics** recibida por **paint** en una referencia **Graphics2D** y la asigna a **g2d** para permitir el acceso a características de Java2D.

La primera figura que dibujamos es una elipse rellena con colores que cambian gradualmente. Las líneas 24 a 29

```

g2d.setPaint(
    new GradientPaint( 5, 30           // x1, y1
                      Color.blue,      // Color inicial
                      35, 100,          // x2, y2
                      Color.yellow,     // fin de Color
                      true ) );        // cíclico

```

invocan al método **setPaint** de **Graphics2D** para establecer el objeto **Paint** que determina el color de la figura a desplegar. Un objeto **Paint** es un objeto de cualquier clase que implementa la interfaz **java.awt.Paint**. El objeto **Paint** puede ser algo tan sencillo como uno de los objetos **Color** predefinidos que presentamos en la sección 28.3 (la clase **Color** implementa a **Paint**), o el objeto **Paint** puede ser una instancia de las clases **GradientPaint**, **SystemColor** o **TexturePaint** de la API Java2D. En este caso, utilizamos un objeto **GradientPaint**.

La clase **GradientPaint** ayuda a dibujar una figura con colores que cambian gradualmente; a lo que se le llama *degradado*. El constructor **GradientPaint** que utilizamos aquí requiere siete argumentos. Los dos primeros argumentos especifican la coordenada inicial del degradado. El tercer argumento especifica el **Color** inicial para el degradado. El cuarto y quinto argumentos especifican la coordenada final del degradado. El sexto argumento especifica el **Color** final del degradado. El último argumento especifica si el degradado es cíclico (**true**) o no cíclico (**false**). Las dos coordenadas determinan la dirección del degradado. La segunda coordenada (*35, 100*) se encuentra abajo y hacia la derecha de la primera coordenada (*5, 30*), por lo que el degradado va hacia abajo y hacia la derecha en un ángulo. Este degradado es cíclico (**true**), por lo que el color comienza en azul, poco a poco se vuelve amarillo, y posteriormente regresa poco a poco al azul. Si el degradado no es cíclico, el color sufre una transición del primer color especificado (por ejemplo, azul) al segundo color (por ejemplo, amarillo).

La línea 30

```
g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
```

utiliza el método **fill** para dibujar un objeto relleno **Shape**. El objeto **Shape** es una instancia de cualquier clase que implementa la interfaz **Shape** (del paquete **java.awt**); en este caso, es una instancia de la clase

Ellipse2D.Double. El constructor **Ellipse2D.Double** recibe cuatro argumentos que especifican el rectángulo que limita la elipse a desplegar.

Después, dibujamos un rectángulo rojo con un borde grueso. La línea 33 utiliza **setPaint** para establecer el objeto **Paint** en **Color.red**. La línea 34

```
g2d.setStroke( new BasicStroke( 10.0f ) );
```

utiliza el método **setStroke** de **Graphics2D** para establecer las características del borde del rectángulo (o las líneas de cualquier otra figura). El método **setStroke** requiere un objeto **Stroke** como su argumento. El objeto **Stroke** es una instancia de cualquier clase que implementa la interfaz **Stroke** (del paquete **java.awt**); en este caso, una instancia de la clase **BasicStroke**. La clase **BasicStroke** proporciona una variedad de constructores para especificar el ancho de la línea, cómo finaliza la línea (llamada *fin de mayúscula*), cómo unir las líneas (llamado *unión de líneas*) y los atributos para puntear la línea (si se trata de una línea punteada). El constructor aquí especifica que la línea debe tener 10 pixeles de ancho.

Las líneas 35 y 36

```
g2d.draw(
    new Rectangle2D.Double( 80, 30, 65, 100 ) );
```

utilizan el método **draw** de **Graphics2D** para dibujar un objeto **Shape**; en este caso una instancia de la clase **Rectangle2D.Double**. El constructor de **Rectangle2D.Double** recibe cuatro argumentos que especifican la coordenada superior izquierda *x*, la coordenada superior izquierda *y*, el ancho y la altura del rectángulo.

Después dibujamos un rectángulo redondeado lleno con un patrón creado en un objeto **BufferedImage** (del paquete **java.awt.image**). Las líneas 39 a 41

```
BufferedImage imagenBuf=
    new BufferedImage(
        10, 10, BufferedImage.TYPE_INT_RGB );
```

crean el objeto **BufferedImage**. La clase **BufferedImage** puede utilizarse para producir imágenes en color y en escala de grises. Este objeto **BufferedImage** es de 10 pixeles de ancho y de 10 pixeles de alto. El tercer argumento del constructor **BufferedImage.TYPE_INT_RGB** indica que la imagen se almacena en color por medio del **esquema de color RGB**.

Para crear el patrón de relleno para el rectángulo redondeado, primero debemos dibujar en el **BufferedImage**. La línea 43

```
Graphics2D gg = imagenBuf.createGraphics();
```

crea un objeto **Graphics2D** que puede utilizarse para dibujar en el **BufferedImage**. Las líneas 44 a 51 utilizan los métodos **setColor**, **fillRect** y **drawRect** (que explicamos anteriormente en este capítulo) para crear el patrón.

Las líneas 54 a 56

```
g2d.setPaint(
    new TexturePaint(
        imagenBuf, new Rectangle( 10, 10 ) ) );
```

establecen el objeto **Paint** en un nuevo objeto **TexturePaint** (del paquete **java.awt**). Un objeto **TexturePaint** utiliza la imagen almacenada en su **BufferedImage** asociada como la textura de relleno para una figura. El segundo argumento especifica el área del **Rectangulo** del **BufferedImage** que se replicará a través de la textura. En este caso, el **Rectangulo** es del mismo tamaño que **BufferedImage**. Sin embargo, puede utilizarse una parte más pequeña de **BufferedImage**.

Las líneas 57 a 59

```
g2d.fill(
    new RoundRectangle2D.Double(
        155, 30, 75, 100, 50, 50 ) );
```

utilizan el método **fill** de **Graphics2D** para dibujar un objeto **Shape** lleno; en este caso, una instancia de la clase **RoundRectangle2D.Double**. El constructor **RoundRectangle2D.Double** recibe seis

argumentos que especifican las dimensiones del rectángulo y el ancho y la altura del arco utilizado para determinar el redondeado de las esquinas.

Después dibujamos un arco en forma de pastel con una línea blanca gruesa. La línea 62 establece el objeto **Paint** en **Color.white**. La línea 63 establece el objeto **Stroke** en un nuevo **BasicStroke** para una línea de 6 pixeles de ancho. Las líneas 64 a 66

```
g2d.draw(
    new Arc2D.Double(
        240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
```

utilizan el método **draw** de **Graphics2D** para dibujar un objeto **Shape**; en este caso un **Arc2D.Double**. Los primeros cuatro argumentos del constructor **Arc2D.Double** especifican la coordenada superior izquierda *x*, la coordenada superior izquierda *y*, el ancho y la altura del rectángulo que limita el arco. El quinto argumento especifica el ángulo inicial. El sexto argumento especifica el ángulo del arco. El último argumento especifica cómo se cierra el arco. La constante **Arc2D.PIE** indica que el arco se cierra dibujando dos líneas; una a partir del punto de inicio del arco hasta el centro del rectángulo delimitador, y otra desde el centro del rectángulo delimitador hasta el punto final. La clase **Arc2D** proporciona otras dos constantes estáticas para especificar cómo cerrar el arco. La constante **Arc2D.CHORD** dibuja una línea desde el punto inicial hasta el punto final. La constante **Arc2D.OPEN** especifica que el arco no está cerrado.

Por último, dibujamos dos líneas utilizando objetos **Line2D**; una continua y otra punteada. La línea 69 establece el objeto **Paint** en **Color.green**. La línea 70

```
g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
```

utiliza el método **draw** de **Graphics2D** para dibujar un objeto **Shape**; en este caso, una instancia de la clase **Line2D.Double**. Los argumentos del constructor **Line2D.Double** especifican las coordenadas iniciales y finales de la línea.

La línea 72 define un arreglo **float** de un elemento que contiene el valor **10**. Este arreglo se utilizará para describir los guiones de la línea punteada. En este caso, cada guión tendrá 10 pixeles de largo. Para crear guiones de diferentes longitudes en un patrón, simplemente proporcione las longitudes de cada uno, como el elemento de un arreglo. La línea 74 establece el objeto **Paint** en **Color.yellow**. Las líneas 75 a 79

```
g2d.setStroke(
    new BasicStroke( 4,
        BasicStroke.CAP_ROUND,
        BasicStroke.JOIN_ROUND,
        10, guiones, 0 ) );
```

establecen el objeto **Stroke** en un nuevo **BasicStroke**. La línea tendrá 4 pixeles de ancho y tendrá bordes redondeados (**BasicStroke.CAP_ROUND**). Si las líneas se unen (como en un rectángulo en las esquinas), la unión de las líneas se redondeará (**BasicStroke.JOIN_ROUND**). El argumento **guiones** especifica las longitudes de los guiones para la línea. El último argumento indica el subíndice de inicio del arreglo **guiones** para el primer guión del patrón. La línea 80 dibuja una línea con el **Stroke** actual.

Un patrón general es una figura construida a partir de líneas rectas y curvas complejas. Un patrón general se representa con un objeto de la clase **GeneralPath** (del paquete **java.awt.geom**). El programa de la figura 28.23 demuestra el dibujo de un patrón general en la figura de una estrella de cinco puntas.

```
1 // Figura 28.23: Figuras2.java
2 // Demostración de un patrón general
3 import javax.swing.*;
4 import java.awt.event.*;
5 import java.awt.*;
6 import java.awt.geom.*;
7
8 public class Figuras2 extends JFrame {
```

Figura 28.23 Demostración de **GeneralPaths** de Java2D. (Parte 1 de 3.)

```
9     public Figuras2()
10    {
11        super( "Dibujando figuras en 2D " );
12
13        setBackground( Color.yellow );
14        setSize( 400, 400 );
15        show();
16    } // fin del constructor Figuras2
17
18    public void paint( Graphics g )
19    {
20        int puntosX[] =
21            { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
22        int puntosY[] =
23            { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
24
25        Graphics2D g2d = ( Graphics2D ) g;
26
27        // crea una estrella a partir de una serie de puntos
28        GeneralPath estrella = new GeneralPath();
29
30        // establece la coordenada inicial del patrón general
31        estrella.moveTo( puntosX[ 0 ], puntosY[ 0 ] );
32
33        // crea la estrella--esto no dibuja la estrella
34        for ( int k = 1; k < puntosX.length; k++ )
35            estrella.lineTo( puntosX[ k ], puntosY[ k ] );
36
37        // cierra la figura
38        estrella.closePath();
39
40        // traslada el origen hacia (200, 200)
41        g2d.translate( 200, 200 );
42
43        // rota alrededor del origen y dibuja estrellas en colores aleatorios
44        for ( int j = 1; j <= 20; j++ ) {
45            g2d.rotate( Math.PI / 10.0 );
46            g2d.setColor(
47                new Color( ( int )( Math.random() * 256 ),
48                            ( int )( Math.random() * 256 ),
49                            ( int )( Math.random() * 256 ) ) );
50            g2d.fill( estrella ); // dibuja una estrella rellena
51        } // fin de for
52    } // fin del método paint
53
54    public static void main( String args[] )
55    {
56        Figuras2 app = new Figuras2();
57
58        app.addWindowListener(
59            new WindowAdapter() {
60                public void windowClosing( WindowEvent e )
61                {
62                    System.exit( 0 );
63                } // fin del método windowClosing
```

Figura 28.23 Demostración de **GeneralPaths** de Java2D. (Parte 2 de 3.)

```

64          } // fin de la clase interna anónima
65      );
66  } // fin de main
67 } // fin de la clase Figuras2

```

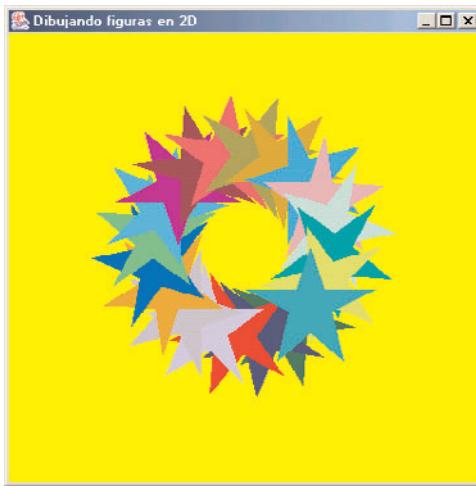


Figura 28.23 Demostración de **GeneralPaths** de Java2D. (Parte 3 de 3.)

Las líneas 20 a 23 definen dos arreglos enteros que representan las coordenadas *x* y *y* de los puntos de la estrella. La línea 28

```
GeneralPath estrella = new GeneralPath();
```

define un objeto **estrella** de **GeneralPath**.

La línea 31

```
estrella.moveTo( PuntosX[ 0 ], PuntosY[ 0 ] );
```

utiliza el método **moveTo** de **GeneralPath** para especificar el primer punto de la estrella. La estructura **for** de las líneas 34 y 35

```
for ( int k = 1; k < PuntosX.length; k++ )
    estrella.lineTo( PuntosX[ k ], PuntosY[ k ] );
```

utilizan el método **lineTo** de **GeneralPath** para dibujar una línea hacia el siguiente punto de la estrella. Cada nueva llamada a **lineTo** dibuja una línea desde el punto anterior al punto actual. La línea 38

```
estrella.closePath();
```

utiliza el método **closePath** de **GeneralPath** para dibujar una línea desde el último punto hasta el punto especificado en la última llamada a **moveTo**. Esto completa el patrón general.

La línea 41

```
g2d.translate( 200, 200 );
```

utiliza el método **translate** de **Graphics2D** para mover el origen del dibujo hacia la posición (200, 200). Todas las operaciones de dibujo ahora utilizan la posición (200, 200) como (0, 0).

La estructura **for** de la línea 44 dibuja 20 veces la **estrella**, rotándola alrededor del punto de origen. La línea 45

```
g2d.rotate( Math.PI / 10.0 );
```

utiliza el método **rotate** de **Graphics2D** para rotar la siguiente figura desplegada. El argumento especifica el ángulo de rotación en radianes (en donde $360^\circ = 2\pi$ radianes). La línea 50 utiliza el método **fill** de **Graphics2D** para dibujar una versión rellena de la **estrella**.

RESUMEN

- Un sistema de coordenadas es un esquema para identificar cada punto posible en la pantalla.
- La esquina superior izquierda de un componente GUI tiene las coordenadas (0,0). Una par de coordenadas se compone de una coordenada *x* (la coordenada horizontal) y una coordenada *y* (la coordenada vertical).
- Las unidades de coordenadas se miden en pixeles. Un píxel es la unidad de resolución más pequeña de un monitor.
- Un contexto gráfico permite dibujar en la pantalla con Java. Un objeto **Graphics** manipula un contexto gráfico al controlar la manera en que se dibuja la información.
- Los objetos **Graphics** contienen métodos para dibujar, para manipular fuentes, para manipular el color, etcétera.
- Por lo general se llama al método **paint** en respuesta a un *evento* tal como el descubrimiento de una ventana.
- El método **repaint** solicita la llamada al método **update** de **Component** lo más pronto posible para limpiar cualquier dibujo previo en el fondo de **Component**, a continuación **update** llama directamente a **paint**.
- La clase **Color** define métodos y constantes para manipular los colores en un programa en Java.
- Java utiliza los colores RGB, en donde el rojo, el verde y el azul son componentes enteros con un rango entre 0 y 255, o valores de punto flotante con un rango entre 0.0 a 1.0. Mientras más grande sea el valor RGB, mayor será la cantidad de un color en particular.
- Los métodos **getRed**, **getGreen** y **getBlue** de **Color** devuelven valores enteros entre 0 y 255 que representan la cantidad de color rojo, verde y azul en un **Color**.
- La clase **Color** proporciona 13 objetos **Color** predefinidos.
- El método **getColor** de **Graphics** devuelve un objeto **Color** que representa el color de dibujo actual. El método **setColor** de **Graphics** establece el contenido actual del color.
- Java proporciona la clase **JColorChooser** para desplegar un cuadro de diálogo para seleccionar colores.
- El método estático **showDialog** de la clase **JColorChooser** despliega el diálogo para la selección de colores. Este método devuelve el objeto **Color** seleccionado (o **null** si no se seleccionó color alguno).
- El diálogo predeterminado **JColorChooser** le permite seleccionar un color entre una variedad de *muestras de colores*. La ficha **HSB** le permite seleccionar un color basándose en el tono, la saturación y el brillo. La ficha **RGB** le permite seleccionar un color con el uso de barras de desplazamiento para los componentes rojo, verde y azul.
- El método **setBackground** de **Component** (uno de los muchos métodos de **Component** que pueden utilizarse en la mayoría de los componentes de un GUI) modifica el color de fondo de un componente.
- El constructor de la clase **Font** toma tres argumentos, el nombre de la fuente, el *estilo de la fuente* y el *tamaño de la fuente*. El nombre de la fuente corresponde a cualquiera que sea soportada por el sistema. El estilo de la fuente es **Font.PLAIN**, **Font.ITALIC** o **Font.BOLD**. El tamaño de la fuente se mide en puntos.
- El método **setFont** de **Graphics** establece la fuente de dibujo.
- La clase **FontMetrics** define varios métodos para obtener la métrica de la fuente.
- El método **getFontMetrics** de **Graphics** sin argumentos obtiene el objeto **FontMetrics** para la fuente actual. El método **getFontMetrics** de **Graphics** que recibe un argumento **Font**, devuelve el objeto **FontMetrics** correspondiente.
- Los métodos **draw3DRect** y **fill3DRect** toman cinco argumentos que especifican la esquina superior izquierda del rectángulo, el ancho y la altura del rectángulo, y si el rectángulo está aumentado (**true**) o disminuido (**false**).
- Los métodos **drawRoundText** y **fillRoundText** dibujan rectángulos con esquinas redondeadas. Sus dos primeros argumentos especifican la esquina superior izquierda, el tercer y cuarto argumentos especifican el **ancho** y la **altura**, y los dos últimos argumentos, **anchoArco** y **alturaArco**, determinan los diámetros horizontal y vertical de los arcos empleados para representar las esquinas.
- Los métodos **drawOval** y **fillOval** toman los mismos argumentos: la coordenada superior izquierda y el ancho y la altura del rectángulo delimitador que contiene la elipse.
- Un arco es una porción de una elipse. Los arcos barren desde un ángulo inicial hasta el número de grados especificado por el ángulo del arco. El ángulo inicial especifica en dónde comienza el arco y el ángulo del arco especifica el número de grados que barre el arco. Los arcos que barren en contra de las manecillas del reloj se miden en grados positivos y los arcos que se barren en favor de las manecillas del reloj se miden en grados negativos.

- Los métodos **drawArc** y **fillArc** toman los mismos argumentos, la coordenada superior izquierda, el **ancho** y la **altura** del rectángulo delimitador que contiene al arco, y **anguloInicial** y **anguloArco** que definen el barrido del arco.
- Los polígonos son figuras de múltiples lados. Las polilíneas son una serie de puntos conectados.
- Un constructor **Polygon** recibe un arreglo que contiene la coordenada *x* para cada punto, un arreglo que contiene la coordenada *y* para cada punto y el número de puntos del polígono.
- Una versión del método **drawPolygon** de **Graphics** despliega un objeto **Polygon**. Otra versión recibe un arreglo que contiene la coordenada *x* de cada punto, un arreglo que contiene la coordenada *y* de cada punto y el número de puntos en el polígono, y despliega el polígono correspondiente.
- El método **drawPolyline** de **Graphics** despliega una serie de líneas conectadas especificada por sus argumentos (un arreglo contiene la coordenada *x* de cada punto, un arreglo que contiene la coordenada *y* de cada punto y el número de puntos).
- El método **addPoint** de **Polygon** agrega pares de coordenadas *x* y *y* a un polígono.
- La API Java2D proporciona capacidades para gráficos de dos dimensiones para el procesamiento de líneas de arte, texto e imágenes.
- Para acceder a las capacidades de **Graphics2D**, convierta el tipo de la referencia **Graphics** que se pasa a **paint** en una referencia a **Graphics2D** como en (**Graphics2D**) *g*.
- El método **setPaint** de **Graphics2D** establece el objeto **Paint** que determina el color y la textura para la figura a desplegar. Un objeto **Paint** es un objeto de cualquier clase que implementa la interfaz de **java.awt.Paint**. El objeto **Paint** puede ser de un **Color** o una instancia de las clases **GradientPaint**, **SystemColor**, o **TexturePaint** de la API Java2D.
- La clase **GradientPaint** dibuja una figura con un color que cambia gradualmente, llamado *degradado*.
- El método **fill** de **Graphics2D** dibuja un objeto **Shape** relleno. El objeto **Shape** es una instancia de cualquier clase que implementa la interfaz **Shape**.
- El constructor **Ellipse2D.Double** recibe cuatro argumentos que especifican el rectángulo que delimita la elipse a desplegar.
- El método **setStroke** de **Graphics2D** establece las características de las líneas utilizadas para dibujar la figura. El método **setStroke** requiere un objeto **Stroke** como su argumento. El objeto **Stroke** es una instancia de cualquier clase que implementa la interfaz **Stroke**, tal como **BasicStroke**.
- El método **draw** de **Graphics2D** dibuja un objeto **Shape**. El objeto **Shape** es una instancia de cualquier clase que implementa la interfaz **Shape**.
- El constructor **Rectangle2D.Double** recibe cuatro argumentos que especifican la coordenada *x* de la esquina superior izquierda, la coordenada *y* de la esquina superior izquierda, el ancho y la altura del rectángulo.
- La clase **BufferedImage** puede utilizarse para producir imágenes en color y en escala de grises.
- Un objeto **TexturePaint** utiliza la imagen almacenada en su objeto **BufferedImage** asociado como la textura de relleno para una figura rellenada.
- El constructor **RoundRectangle2D.Double** recibe seis argumentos que especifican las dimensiones del rectángulo, y el ancho y la altura del arco para determinar las esquinas redondeadas.
- Los cuatro primeros argumentos del constructor **Arc2D.Double** especifican la coordenada *x* de la esquina superior izquierda, la coordenada *y* de la esquina superior izquierda para el arco. El quinto argumento especifica el ángulo inicial. El sexto argumento especifica el ángulo final. El último argumento especifica el tipo del arco (**Arc2D**, **PIE**, **Arc2D**, **CHORD** o **Arc2D**, **OPEN**).
- Los argumentos del constructor **Line2D.Double** especifican las coordenadas inicial y final de la línea.
- Un patrón general es una figura construida a partir de líneas rectas y curvas complejas representadas mediante un objeto de la clase **GeneralPath** (del paquete **java.awt.geom**).
- El método **moveTo** de **GeneralPath** especifica el primer punto del patrón general. El método **lineTo** de **GeneralPath** dibuja una línea hacia el siguiente punto del patrón general. Cada nueva llamada a **lineTo** dibuja una línea desde el punto previo al punto actual. El método **closePath** de **GeneralPath** dibuja una línea desde el último punto hasta el punto especificado en la última llamada a **moveTo**.
- El método **translate** de **Graphics2D** mueve el origen del dibujo hacia una nueva posición. Todas las operaciones de dibujo ahora utilizan la posición como (0,0).

TERMINOLOGÍA

altura del arco	estilo de la fuente	método getFont
ancho del arco	evento	método getFontList
ángulo	fuente	método getFontMetrics
API Java2D	fuente Monospaced	método getGreen
arco limitado por un rectángulo	fuente SansSerif	método getHeight
ascendente	fuente Serif	método getLeading
barrido de un arco	grado	método getName
clase Arc2D.Double	grados negativos	método getRed
clase BufferedImage	grados positivos	método getSize
clase Color	interfaz Paint	método getStyle
clase Componente	interfaz Shape	método isBold
clase Ellipse2D.Double	interfaz Stroke	método isItalic
clase Font	interlineado	método isPlain
clase FontMetrics	línea base	método lineTo
clase GeneralPath	método addPoint	método moveTo
clase GradientPaint	método closePath	método paint
clase Graphics	método draw	método repaint
clase Graphics2D	método draw3DRect	método setColor
clase Line2D.Double	método drawArc	método setFont
clase Polygon	método drawLine	método setPaint
clase Rectangle2D.	método drawOval	método setStroke
Double	método drawPolygon	método translate
clase RoundRectangle2D.	método drawPolyline	método update
Double	método drawRect	métrica de la fuente
clase SystemColor	método drawRoundRect	nombre de la fuente
clase TexturePaint	método fill	objeto gráfico
color de fondo	método fill3DRect	píxel
componente vertical	método fillArc	polígono
contexto gráfico	método fillOval	polígono cerrado
coordenada	método fillPolygon	polígono relleno
coordenada <i>x</i>	método fillRect	proceso controlado por
coordenada <i>y</i>	método fillRoundRect	eventos
descendente	método getAscent	punto
dibujar un arco	método getBlue	rectángulo delimitador
eje <i>x</i>	método getDescent	sistema de coordenadas
eje <i>y</i>	método getFamily	valor RGB

ERRORES COMUNES DE PROGRAMACIÓN

- 28.1** Escribir cualquier constante estática de clase de **Color** con una letra mayúscula inicial, es un error de sintaxis.
- 28.2** Especificar una fuente que no está disponible en un sistema, es un error lógico. Java sustituirá a la fuente predeterminada por el sistema.
- 28.3** Si el número de puntos especificados en el tercer argumento del método **drawPolygon** o del método **fillPolygon** es mayor que el número de elementos de los arreglos de coordenadas que definen el polígono a desplegar, se lanza una **ArrayIndexOutOfBoundsException**.

TIPS DE PORTABILIDAD

- 28.1** Diferentes pantallas tienen diferentes resoluciones (es decir, varía la densidad de pixeles). Esto puede provocar que los gráficos parezcan de tamaño diferente en diferentes pantallas.
- 28.2** El número de fuentes varía mucho a través de los sistemas. El JDK garantiza que las fuentes **Serif**, **Monospaced**, **SansSerif**, **Dialog** y **DialogInput** estarán disponibles.
- 28.3** Java utiliza nombres de fuentes estandarizados y los mapea en sistemas específicos de nombres de fuentes para portabilidad. Esto es transparente para el programador.

OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 28.1** La coordenada superior izquierda (0,0) de una ventana en realidad se encuentra debajo de la barra de título de la ventana. Por esta razón, las coordenadas de dibujo deben ajustarse para dibujar dentro de los bordes de la ventana. La clase **Container** (una superclase de todas las ventanas en Java) contiene el método **getInsets** que devuelve un objeto **Insets** (del paquete **java.awt**) para este propósito. Un objeto **Insets** contiene cuatro miembros públicos, **top**, **bottom**, **left** y **right**, que representan el número de píxeles de cada borde de la ventana hacia el área de dibujo de ésta.
- 28.2** Para modificar el color, usted debe crear un objeto **Color** (o utilizar una de las constantes predefinidas de **Color**); no existen métodos **set** (establecer) en la clase **Color** para modificar las características del color actual.
- 28.3** Para modificar la fuente, debe crear un nuevo objeto **Font**; no existen métodos establecer (set) en la clase **Font** para modificar las características de la fuente actual.

EJERCICIOS DE AUTOEVALUACIÓN

- 28.1** Complete los espacios en blanco:
- En Java2D, el método _____ de la clase _____ establece las características de una línea que se utiliza para dibujar una línea.
 - La clase _____ ayuda a definir el relleno para una figura que cambia gradualmente de un color a otro.
 - El método _____ de la clase **Graphics** dibuja líneas entre dos puntos.
 - RGB son las iniciales en inglés para _____, _____ y _____.
 - Los tamaños de las fuentes se miden en unidades llamadas _____.
 - La clase _____ ayuda a definir el relleno para una figura que utiliza un patrón dibujado dentro de un objeto de la clase **BufferedImage**.
- 28.2** Establezca si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- Los primeros dos argumentos del método **drawOval** de **Graphics** especifican las coordenadas del centro de la elipse.
 - En el sistema de coordenadas de Java, los valores de *x* se incrementan de izquierda a derecha.
 - El método **fillPolygon** dibuja un polígono sólido con el color actual.
 - El método **drawArc** permite ángulos negativos.
 - El método **getSize** devuelve el tamaño de la fuente actual en centímetros.
 - La coordenada de píxel (0,0) se localiza exactamente en el centro del monitor.
- 28.3** Encuentre el/los error(es) en cada una de las siguientes instrucciones y explique cómo corregirlos. Asuma que **g** es un objeto de **Graphics**.
- g.setFont("SansSerif");**
 - g.erase(x, y, a, h); // limpia el rectángulo en (x, y)**
 - Font f = new Font("Serif", Font.BOLDITALIC, 12);**
 - g.setColor (Color.Yellow); // cambia el color a amarillo**

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

- 28.1** a) **setStroke**, **Graphics2D**. b) **GradientPaint**. c) **drawLine**. d) Red, green, blue. e) Puntos. f) **TexturePaint**.
- 28.2** a) Falso. Los dos primeros argumentos especifican la esquina superior izquierda del rectángulo delimitador.
 b) Verdadero.
 c) Verdadero.
 d) Verdadero.
 e) Falso. Los tamaños de las fuentes se miden en puntos.
 f) Falso. La coordenada (0,0) corresponde a la esquina superior izquierda de un componente GUI en el cual ocurre el dibujo.
- 28.3** a) El método **setFont** toma un objeto **Font** como argumento, no una cadena.
 b) La clase **Graphics** no contiene un método **erase**. Se debe utilizar el método **clearRect**.
 c) **Font.BOLDITALIC** no es un estilo de fuente válido. Para obtener una fuente en negritas y cursivas, utilice **Font.BOLD + Font.ITALIC**.
 d) **Yellow** de comenzar con una letra minúscula: **g.setColor(Color.yellow);**

EJERCICIOS

- 28.4** Complete los espacios en blanco:
- La clase _____ de la API Java2D se utiliza para definir elipses.
 - Los métodos **draw** y **fill** de la clase **Graphics2D** requieren un objeto de tipo _____ como argumento.
 - Las tres constantes que especifican el tipo de fuente son _____, _____ y _____.
 - El método _____ de **Graphics2D** establece el color de pintura para las figuras de Java2D.
- 28.5** Establezca si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- El método **drawPolygon** conecta automáticamente los puntos de finalización del polígono.
 - El método **drawLine** dibuja una línea entre dos puntos.
 - El método **fillArc** utiliza grados para especificar un ángulo.
 - En el sistema de coordenadas de Java, los valores de y se incrementan de abajo hacia arriba.
 - La clase **Graphics** hereda directamente de la clase **Object**.
 - La clase **Graphics** es una clase **abstract**.
 - La clase **Font** hereda directamente desde la clase **Graphics**.
- 28.6** Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deben estar separados entre sí por 10 pixeles. Utilice el método **drawOval** de la clase **Graphics**.
- 28.7** Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deben estar separados entre sí por 10 pixeles. Utilice el método **drawArc**.
- 28.8** Modifique su solución del ejercicio 28.6 para dibujar elipses por medio de instancias de la clase **Ellipse2D**. **Double** y del método **draw** de la clase **Graphics2D**.
- 28.9** Escriba un programa que dibuje líneas con longitudes y colores aleatorios.
- 28.10** Modifique su solución del ejercicio 28.9 para dibujar líneas aleatorias, con colores y gruesos de línea aleatorios. Utilice la clase **Line2D.Double** y el método **draw** de la clase **Graphics2D** para dibujar las líneas.
- 28.11** Escriba un programa que despliegue triángulos generados de manera aleatoria con colores diferentes. Cada triángulo debe llenarse con un color diferente. Utilice la clase **GeneralPath** y el método **fill** de la clase **Graphics2D** para dibujar los triángulos.
- 28.12** Escriba un programa que dibuje de manera aleatoria caracteres de diferentes tamaños y colores.
- 28.13** Escriba un programa que dibuje una rejilla de 8 por 8. Utilice el método **drawLine**.
- 28.14** Modifique su solución del ejercicio 28.13 para dibujar una rejilla por medio de instancias de la clase **Line2D**. **Double** y el método **draw** de la clase **Graphics2D**.
- 28.15** Escriba un programa que dibuje una rejilla de 10 por 10. Utilice el método **drawRect**.
- 28.16** Modifique su solución al ejercicio 28.15 para dibujar la rejilla por medio de instancias de la clase **Rectangle2D**. **Double** y del método **draw** de la clase **Graphics2D**.
- 28.17** Escriba un programa que dibuje un tetraedro (una pirámide). Utilice la clase **GeneralPath** y el método **draw** de la clase **Graphics2D**.
- 28.18** Escriba un programa que dibuje un cubo. Utilice la clase **GeneralPath** y el método **draw** de la clase **Graphics2D**.
- 28.19** Escriba una aplicación que simule un protector de pantalla. La aplicación debe dibujar líneas de manera aleatoria con el uso del método **drawLine** de la clase **Graphics**. Después de dibujar 100 líneas, la aplicación debe limpiarse a sí misma y comenzar a dibujar las líneas de nuevo. Para permitir que el programa dibuje de manera continua, coloque una llamada a **repaint** como la última línea del método **paint**. ¿Nota usted algún problema con esto en su sistema?
- 28.20** Aquí tenemos un poco más. El paquete **javax.swing** contiene una clase llamada **Timer** que es capaz de llamar al método **actionPerformed** de la interfaz **ActionListener** en un intervalo fijo de tiempo (especificado en milisegundos). Modifique la solución del ejercicio 28.19 para eliminar la llamada a **repaint** desde el método **paint**. Defina su clase de modo que implemente **ActionListener** (el método **actionPerformed** simplemente debe llamar a **repaint**). Defina una variable de instancia de tipo **Timer** llamada **crono** en su clase. En el constructor para su clase, escriba las siguientes instrucciones:

```
crono = new Timer( 1000, this );
crono.start();
```

Esto crea una instancia de la clase **Timer** que llamará al objeto **actionPerformed** del objeto **this** cada 1000 milisegundos (es decir, cada segundo).

- 28.21** Modifique su solución del ejercicio 28.20 para permitir al usuario escribir un número de líneas aleatorias que deben dibujarse antes de que la aplicación se limpie a sí misma y comience a dibujar de nuevo las líneas. Utilice **JTextField** para obtener el valor. El usuario debe poder escribir un nuevo número dentro de **JTextField** en cualquier momento durante la ejecución del programa. [Nota: Combinar los componentes Swing del GUI y las guías de dibujo pueden provocar problemas interesantes para los cuales le presentamos soluciones en el capítulo 29.] Por ahora, la primera línea del método **paint** debe ser

```
super.paint( g );
```

para garantizar que los componentes GUI se desplieguen apropiadamente. Usted notará que algunas de las líneas dibujadas de manera aleatoria oscurecerán el **JTextField**. Utilice una definición interna de la clase para realizar la manipulación de eventos para **JTextField**.

- 28.22** Modifique su solución al ejercicio 28.20 para elegir de manera aleatoria diferentes figuras a desplegar (utilice los métodos de la clase **Graphics**).
- 28.23** Modifique su solución del ejercicio 28.22 para utilizar las clases y las capacidades de dibujo de la API Java2D. Para figuras tales como rectángulos y elipses, dibújelas con degradados generados de manera aleatoria (utilice la clase **GradientPaint** para generar el degradado).
- 28.24** Escriba un programa que utilice el método **drawPolyline** para dibujar una espiral.
- 28.25** Escriba un programa que introduzca cuatro números y que grafique dichos números en una gráfica de pastel. Utilice la clase **Arc2D.Double** y el método **fill** de la clase **Graphics2D** para realizar el dibujo. Dibuje cada pieza del pastel con un color diferente.
- 28.26** Escriba un applet que introduzca cuatro números y que grafique dichos números en una gráfica de barra. Utilice la clase **Rectangle2D.Double** y el método **fill** de la clase **Graphics2D** para realizar el dibujo. Dibuje cada barra con un color diferente.

Componentes de la interfaz gráfica de usuario de Java

Objetivos

- Comprender los principios de diseño de las interfaces gráficas de usuario.
- Crear interfaces gráficas de usuario.
- Comprender los paquetes que contienen componentes de interfaces gráficas de usuario y clases e interfaces para manejo de eventos.
- Crear y manipular botones, etiquetas, listas, campos de texto y paneles.
- Comprender los eventos del ratón y del teclado.
- Comprender y utilizar los administradores de diseño.

... los profetas más sabios primero se aseguran del evento.

Horace Walpole

¿Crees que puedo escuchar estas cosas durante todo el día?

Lewis Carroll

Habla en afirmativo; enfatiza tu elección ignorando totalmente todo lo que rechazas.

Ralph Waldo Emerson

Paga con tu dinero y toma tus propias decisiones.

Punch

Adivina si puedes, elige si te atreves.

Pierre Corneille

¡Todo aquel que entra aquí, pierde toda esperanza!

Dante Alighieri



Plan general

- 29.1 Introducción**
- 29.2 Generalidades de Swing**
- 29.3 JLabel**
- 29.4 Modelo de manejo de eventos**
- 29.5 JTextField y JPasswordField**
 - 29.5.1 Cómo funciona el manejo de eventos**
- 29.6 JTextArea**
- 29.7 JButton**
- 29.8 JCheckBox**
- 29.9 JComboBox**
- 29.10 Manejo de eventos del ratón**
- 29.11 Administradores de diseño**
 - 29.11.1 FlowLayout**
 - 29.11.2 BorderLayout**
 - 29.11.3 GridLayout**
- 29.12 Paneles**
- 29.13 Creación de una subclase autocontenido de JPanel**
- 29.14 Ventanas**
- 29.15 Uso de menús con marcos**

Resumen • Terminología • Errores comunes de programación • Buenas prácticas de programación • Observaciones de apariencia visual • Tip de portabilidad • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios

29.1 Introducción

Una *interfaz gráfica de usuario (GUI)* muestra una interfaz ilustrada de un programa. Una GUI proporciona una “apariencia visual” única a un programa. Ya que las GUIs ofrecen a las diversas aplicaciones un conjunto consistente de componentes intuitivos de interfaz de usuario, el usuario no necesita invertir tanto tiempo en recordar qué es lo que hacen las secuencias de teclas y puede utilizar el programa de forma más productiva.



Observación de apariencia visual 29.1
Las interfaces de usuario consistentes permiten a un usuario aprender a utilizar nuevas aplicaciones en menos tiempo.

Como ejemplo de una GUI, la figura 29.1 contiene una ventana del Internet Explorer con algunos de sus componentes GUI etiquetados. En esta ventana hay una *barra de menús*, la cual contiene menús (**Archivo**, **Edición**, **Ver**, etcétera). Debajo de la barra de menús hay un conjunto de *botones*, cada uno de los cuales tiene una tarea definida en Internet Explorer. Debajo de los botones hay un *campo de texto*, en el que el usuario puede escribir el nombre de un sitio de la World Wide Web que desee visitar. A la izquierda del campo de texto hay una *etiqueta* que indica cuál es el propósito de este campo. Los menús, botones, campos de texto y etiquetas forman parte de la GUI del Internet Explorer. Éstos le permiten interactuar con el programa. En éste y en el siguiente capítulo, mostraremos estos componentes GUI.

Las GUIs se crean a partir de *componentes GUI* (a los que algunas veces se les llama *controles* o “*widgets*”: una abreviatura de *accesorios de ventana*). Un componente GUI es un objeto con el que el usuario interactúa mediante el ratón o el teclado. En la figura 29.2 aparecen varios componentes GUI comunes. En las siguientes secciones hablaremos detalladamente sobre cada uno de estos componentes GUI. En el siguiente capítulo hablaremos sobre componentes GUI más avanzados.



Figura 29.1 Una ventana de ejemplo del Internet Explorer con los componentes GUI.

Componente	Descripción
JLabel	Un área en la que pueden desplegarse iconos o texto que no puede editarse.
JTextField	Un área en la que el usuario introduce datos a través del teclado. Esta área también puede desplegar información.
JButton	Un área que desencadena un evento, cuando se hace clic sobre ella.
JCheckBox	Un componente GUI que puede estar, o no, seleccionado.
JComboBox	Una lista desplegable de elementos que el usuario puede seleccionar, haciendo clic en un elemento de la lista o escribiendo en el cuadro, si esto está permitido.
JList	Un área en la que aparece una lista de elementos que el usuario puede seleccionar, haciendo clic una vez en cualquier elemento de la lista. Si hace doble clic en un elemento de la lista, generará un evento de acción. Es posible seleccionar varios elementos.
Jpanel	Un contenedor en el que pueden colocarse otros componentes.

Figura 29.2 Algunos componentes GUI básicos.

29.2 Generalidades de Swing

Las clases que se utilizan para crear los componentes GUI de la figura 29.2 forman parte de los *componentes GUI de Swing*, que se encuentran en el paquete **javax.swing**. Éstos son los componentes GUI más recientes de la plataforma Java 2. Los *componentes Swing* (como se les llama comúnmente) están escritos, se manipulan y se despliegan completamente en Java (por lo cual se les llama *componentes puros de Java*).

Los componentes GUI originales del paquete *Abstract Windowing Toolkit*, **java.awt** (también conocido como *AWT*) están enlazados directamente a las herramientas de la interfaz gráfica de usuario de la plataforma local. Por lo tanto, un programa en Java que se ejecuta en distintas plataformas Java tiene una apariencia distinta, e incluso, algunas veces hasta las interacciones del usuario son distintas en cada plataforma. Juntas, a la apariencia y a la forma en que el usuario interactúa con el programa se les conoce como la *apariencia visual* del programa. Los componentes Swing permiten al programador especificar una apariencia visual distinta para cada plataforma, una apariencia visual uniforme entre todas las plataformas, o incluso puede cambiar la apariencia visual mientras el programa se ejecuta.

Observación de apariencia visual 29.2



Los componentes Swing están escritos en Java, por lo que ofrecen un mayor nivel de portabilidad y flexibilidad que los componentes GUI originales de Java del paquete **java.awt**.

Los componentes Swing se conocen comúnmente como *componentes ligeros*; están escritos completamente en Java, por lo que no les afectan las complejas herramientas GUI de la plataforma en la que se utilizan. A los componentes AWT (muchos de los cuales son comparables con los componentes Swing) que se enlazan a la plataforma local se les llama *componentes pesados*; éstos dependen del *sistema de ventanas* de la plataforma local para determinar su funcionalidad y su apariencia visual. Cada componente pesado tiene un *componente asociado* (del paquete `java.awt.peer`), el cual es responsable de las interacciones entre el componente pesado y la plataforma local para mostrarlo y manipularlo. Varios componentes Swing siguen siendo componentes pesados. En particular, las subclases de `java.awt.Window` (como la subclase `JFrame` que utilizamos en capítulos anteriores) que muestran ventanas en la pantalla, aún requieren de una interacción directa con el sistema de ventanas local. Como tales, los componentes GUI pesados de Swing son menos flexibles que muchos de los componentes ligeros que presentaremos.

Tip de portabilidad 29.1



La apariencia de una GUI definida con componentes GUI pesados del paquete `java.awt` puede variar entre plataformas. Los componentes pesados se “enlazan” a la GUI de la plataforma “local”, la cual varía entre las distintas plataformas.

La figura 29.3 muestra una jerarquía de herencia de las clases que definen los atributos y comportamientos comunes para la mayoría de los componentes Swing. Cada clase aparece con su nombre y con el nombre completo de su paquete. La mayor parte de la funcionalidad de cada componente GUI se deriva de esas clases. Una clase que hereda de la clase `Component` es un componente. Por ejemplo, la clase `Container` hereda de la clase `Component`, y ésta hereda de `Object`. Por lo tanto, un objeto `Container` es un objeto `Component` y también es un `Object`, y un objeto `Component` es un `Object`. Una clase que hereda de la clase `Container` es un `Container`. Por lo tanto, un objeto `JComponent` es un `Container`.

Observación de ingeniería de software 29.1



Para utilizar componentes GUI con efectividad debe comprender las jerarquías de herencia de `javax.swing` y `java.awt`; en especial de las clases `Component`, `Container` y `JComponent`, que definen características comunes para la mayoría de los componentes Swing.

La clase `Component` define los métodos que pueden aplicarse a un objeto de cualquier subclase de `Component`. Dos de los métodos que se originan en la clase `Component` los hemos utilizado con frecuencia hasta este punto del texto: `paint` y `repaint`. Es importante comprender los métodos de la clase `Component`, ya que la mayor parte de la funcionalidad heredada por cada una de las subclases de `Component` está originalmente definida por la clase `Component`. Las operaciones comunes a la mayoría de los componentes GUI (tanto Swing como AWT) se encuentran en la clase `Component`.

Buena práctica de programación 29.1



Estudie los métodos de la clase `Component` que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de la mayoría de los componentes GUI.

Un objeto `Container` es una colección de componentes relacionados. En aplicaciones con objetos `JFrame` y en applets, adjuntamos componentes al panel de contenido: un objeto `Container`. La clase `Container` define el conjunto de métodos que pueden aplicarse a un objeto de cualquier subclase de `Container`. Uno de los métodos que se origina en la clase `Container`, y que hemos utilizado frecuentemente hasta este punto del texto, para agregar componentes a un panel de contenido, es `add`. Otro método que se origina en la

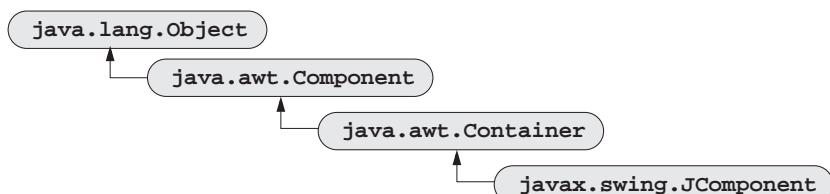


Figura 29.3 Superclases comunes de muchos de los componentes Swing.

clase **Container** es **setLayout**, el cual hemos utilizado para especificar el administrador de diseño que ayuda a un objeto **Container** a posicionar y ajustar el tamaño de sus componentes.



Buena práctica de programación 29.2

*Estudie los métodos de la clase **Container** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.*

La clase **JComponent** es la superclase de la mayoría de los componentes Swing. Esta clase define el conjunto de métodos que pueden aplicarse a un objeto de cualquier subclase de **JComponent**.



Buena práctica de programación 29.3

*Estudie los métodos de la clase **JComponent** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.*

Los componentes Swing que corresponden a subclases de **JComponent** tienen muchas características, las cuales incluyen:

1. Una *apariencia visual adaptable* que puede utilizarse para personalizar la apariencia visual cuando el programa se ejecuta en distintas plataformas.
2. Teclas de acceso directo (llamadas *mnemónicos*) para acceder directamente a los componentes GUI a través del teclado.
3. Herramientas para manejo de eventos comunes, para casos en los que varios componentes GUI inicien las mismas acciones en un programa.
4. Breves descripciones del propósito de un componente GUI (que se conocen como *cuadros de información de herramientas*) que aparecen cuando el cursor del ratón se posiciona sobre el componente durante un lapso de tiempo corto.
5. Soporte para tecnologías de asistencia tales como los lectores de pantalla braille para personas ciegas.
6. Soporte para *localización* de la interfaz de usuario; es decir, para personalizar la interfaz de usuario de manera que aparezca en distintos lenguajes y convenciones culturales.

Éstas son sólo algunas de las muchas características de los componentes Swing. En el resto de este capítulo hablaremos sobre varias de estas características.

29.3 JLabel

Las etiquetas proporcionan instrucciones de texto o información en una GUI. Éstas se definen mediante la clase **JLabel**; una subclase de **JComponent**. Una etiqueta muestra una sola línea de *texto de sólo lectura*. Una vez creadas las etiquetas, los programas raras veces cambian el contenido de una etiqueta. La aplicación de la figura 29.4 muestra el uso de **JLabel**.

```

1 // Figura 29.4: PruebaEtiqueta.java
2 // Demostración de la clase JLabel.
3 import javax.swing.*;
4 import java.awt.*;
5 import java.awt.event.*;
6
7 public class PruebaEtiqueta extends JFrame {
8     private JLabel etiqueta1, etiqueta2, etiqueta3;
9
10    public PruebaEtiqueta()
11    {
12        super( "Prueba de JLabel" );
13
14        Container c = getContentPane();

```

Figura 29.4 Demostración de la clase **JLabel**; **PruebaEtiqueta.java**. (Parte 1 de 2.)

```

15    c.setLayout( new FlowLayout() );
16
17    // Constructor de JLabel con un argumento tipo cadena
18    etiqueta1 = new JLabel( "Etiqueta con texto" );
19    etiqueta1.setToolTipText( "Esta es la etiqueta1" );
20    c.add( etiqueta1 );
21
22    // Constructor de JLabel con argumentos tipo cadena, Icon
23    // y de alineación
24    Icon insecto = new ImageIcon( "insecto1.gif" );
25    etiqueta2 = new JLabel( "Etiqueta con texto e icono",
26                           insecto, SwingConstants.LEFT );
27    etiqueta2.setToolTipText( "Esta es la etiqueta2" );
28    c.add( etiqueta2 );
29
30    // Constructor de JLabel sin argumentos
31    etiqueta3 = new JLabel();
32    etiqueta3.setText( "Etiqueta con icono y texto en la parte inferior" );
33    etiqueta3.setIcon( insecto );
34    etiqueta3.setHorizontalTextPosition(
35        SwingConstants.CENTER );
36    etiqueta3.setVerticalTextPosition(
37        SwingConstants.BOTTOM );
38    etiqueta3.setToolTipText( "Esta es la etiqueta3" );
39    c.add( etiqueta3 );
40
41    setSize( 275, 170 );
42    show();
43 } // fin del constructor PruebaEtiqueta
44
45 public static void main( String args[] )
46 {
47     PruebaEtiqueta ap = new PruebaEtiqueta();
48
49     ap.addWindowListener(
50         new WindowAdapter() {
51             public void windowClosing( WindowEvent e )
52             {
53                 System.exit( 0 );
54             } // fin del método windowClosing
55         } // fin de la clase interna anónima
56     ); // fin de addWindowListener
57 } // fin de main
58 } // fin de la clase PruebaEtiqueta

```

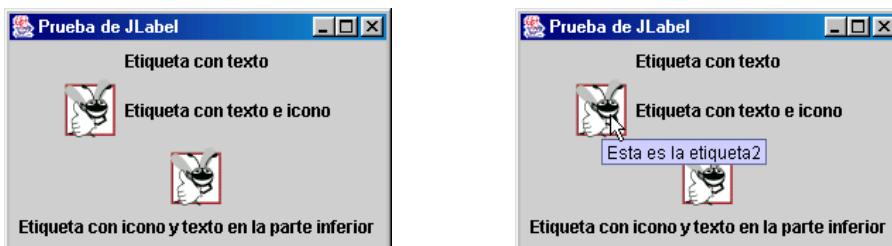


Figura 29.4 Demostración de la clase **JLabel**; **PruebaEtiqueta.java**. (Parte 2 de 2.)



Buena práctica de programación 29.4

Estudie los métodos de la clase `javax.swing.JLabel` que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas completas de la clase antes de usarla.

El programa declara tres referencias `JLabel` en la línea 8:

```
private JLabel etiqueta1, etiqueta2, etiqueta3;
```

Los objetos `JLabel` se instancian en el constructor (línea 10). La instrucción

```
etiqueta1 = new JLabel( "Etiqueta con texto" );
```

de la línea 18 crea un objeto `JLabel` con el texto "`Etiqueta con texto`". El texto se despliega en la etiqueta. La línea 19

```
etiqueta1.setToolTipText( "Esta es la etiqueta1" );
```

utiliza el método `setToolTipText` (heredado por la clase `JLabel` de la clase `JComponent`) para especificar la información de la herramienta (vea la captura de pantalla del lado derecho de la figura 29.4) que aparece cuando el usuario coloca el cursor del ratón sobre la etiqueta de la GUI. Cuando ejecute este programa, intente posicionar el ratón sobre cada una de las etiquetas para ver la información de su herramienta. En la línea 20 se agrega `etiqueta1` al panel de contenido.



Observación de apariencia visual 29.3

Utilice los cuadros de información de herramienta (establecidos mediante el método `setToolTipText` de `JComponent`) para agregar texto descriptivo a sus componentes GUI. Este texto ayuda al usuario a determinar el propósito del componente GUI en la interfaz de usuario.

Muchos componentes Swing pueden mostrar imágenes especificando un objeto `Icon` como argumento para su constructor, o utilizando un método que generalmente se llama `setIcon`. Un objeto `Icon` es un objeto de cualquier clase que implementa la interfaz `Icon` (del paquete `javax.swing`). Una de esas clases es `ImageIcon` (del paquete `javax.swing`), la cual soporta dos formatos de imagen: *GIF* (*Formato de intercambio de gráficos*) y *JPEG* (*Grupo unido de expertos en fotografía*). Los nombres de archivo para cada uno de estos tipos terminan generalmente con `.gif` o `.jpg` (o `.jpeg`), respectivamente.

En el capítulo 30, hablaremos sobre las imágenes con más detalle. La línea 24:

```
Icon insecto = new ImageIcon( "insecto1.gif" );
```

define un objeto `ImageIcon`. El archivo `insecto1.gif` contiene la imagen a cargar y a guardar en el objeto `ImageIcon`. Este archivo debe encontrarse en el mismo directorio que el programa (en el capítulo 30 veremos cómo puede colocarse el archivo en cualquier otra parte). El objeto `ImageIcon` se asigna a la referencia `Icon` llamada `insecto`. Recuerde que la clase `ImageIcon` implementa la interfaz `Icon`, por lo tanto, un objeto `ImageIcon` es un `Icon`.

La clase `JLabel` soporta el despliegue de objetos `Icon`. Las líneas 25 y 26:

```
etiqueta2 = new JLabel( "Etiqueta con texto e icono" );
insecto, SwingConstants.LEFT );
```

utilizan otro constructor de `JLabel` para crear una etiqueta que muestre el texto "`Etiqueta con texto e icono`", y el objeto `Icon` al cual `insecto` hace referencia, y se *justifica o alinea hacia la izquierda* (es decir, el icono y el texto se encuentran en la parte izquierda del área de la etiqueta en la pantalla). La interfaz `SwingConstants` (del paquete `javax.swing`) define un conjunto de constantes enteras comunes (como `SwingConstants.LEFT`), las cuales se utilizan con muchos componentes Swing. De manera predeterminada, el texto aparece a la derecha de la imagen cuando una etiqueta contiene texto y una imagen. Las alineaciones horizontal y vertical de una etiqueta pueden establecerse mediante los métodos `setHorizontalAlignment` y `setVerticalAlignment`, respectivamente. La línea 27 especifica el texto de la información de la herramienta para la `etiqueta2`. En la línea 28 se agrega `etiqueta2` al panel de contenido.



Error común de programación 29.1

Olvidar agregar un componente a un contenedor, para que pueda mostrarse en pantalla, es un error lógico en tiempo de ejecución.



Error común de programación 29.2

*Si se agrega a un contenedor un componente que no se haya instanciado, se lanza una excepción **NullPointerException**.*

La clase **JLabel** proporciona muchos métodos para configurar una etiqueta, después de que se ha instanciado. En la línea 31 se crea un objeto **JLabel** y se invoca el constructor sin argumentos (el predeterminado). Dicha etiqueta no tiene texto ni objeto **Icon**. La línea 32

```
etiqueta3.setText( "Etiqueta con icono y texto en la parte inferior" );
```

utiliza el método **setText** de **JLabel** para establecer el texto que aparece en la etiqueta. Su método **getText** correspondiente recupera el texto actual desplegado en una etiqueta. La línea 33

```
etiqueta3.setIcon( insecto );
```

utiliza el método **setIcon** de **JLabel** para establecer el objeto **Icon** que aparece en la etiqueta. Su método **getIcon** correspondiente recupera el objeto **Icon** actual desplegado en una etiqueta. Las líneas 34 a 37

```
etiqueta3.setHorizontalTextPosition(  
    SwingConstants.CENTER );  
etiqueta3.setVerticalTextPosition(  
    SwingConstants.BOTTOM );
```

utilizan los métodos **setHorizontalTextPosition** y **setVerticalTextPosition** de **JLabel** para especificar la posición del texto en la etiqueta. Las instrucciones anteriores indican que el texto se centrará horizontalmente y aparecerá en la parte inferior de la etiqueta. Por lo tanto, el objeto **Icon** aparecerá encima del texto. La línea 38 especifica el texto de la información de la herramienta para la **etiqueta3**. La línea 39 agrega **etiqueta3** al panel de contenido.

29.4 Modelo de manejo de eventos

En la sección anterior no hablamos sobre el manejo de eventos, ya que no hay eventos específicos para los objetos **JLabel**. Las GUIs están *controladas por eventos* (es decir, generan *eventos* cuando el usuario del programa interactúa con la GUI). Algunas interacciones comunes son mover el ratón, hacer clic con el ratón, hacer clic en un botón, escribir en un campo de texto, seleccionar un elemento de un menú, cerrar una ventana, etcétera. Siempre que ocurre una interacción con el usuario, se envía un evento al programa. La información de los eventos de la GUI se almacena en un objeto de una clase que extiende a **AWTEvent**. La figura 29.5 muestra una jerarquía que contiene muchas de las clases de eventos que utilizamos del paquete **java.awt.event**. Muchas de estas clases de eventos las describiremos a lo largo de este capítulo. Los tipos de eventos del paquete

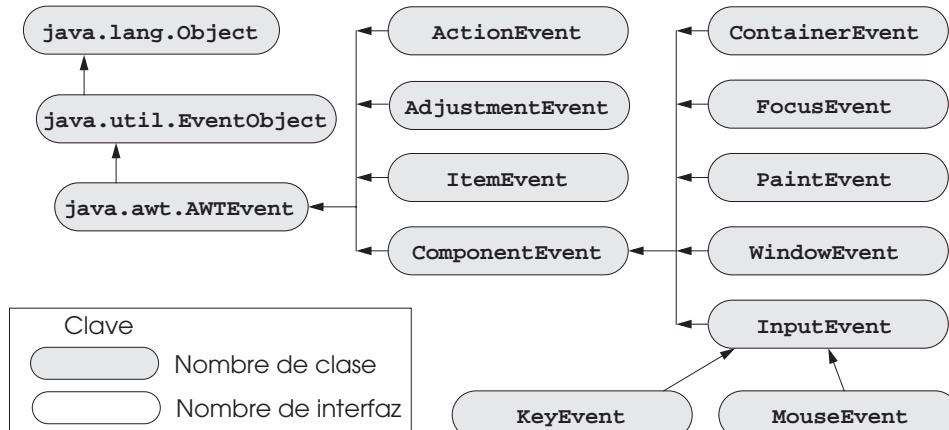


Figura 29.5 Algunas clases de eventos del paquete **java.awt.event**.

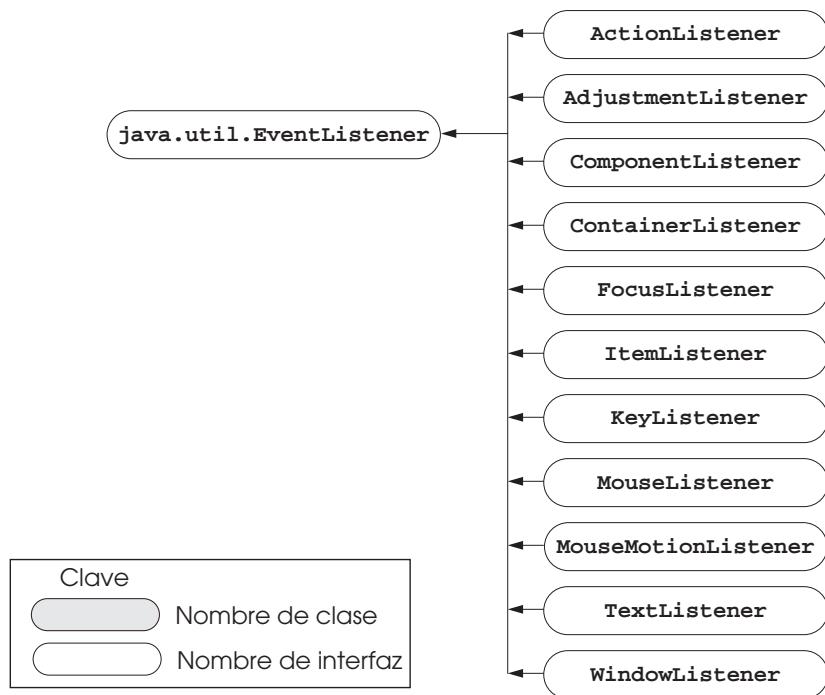


Figura 29.6 Interfaces que escuchan eventos del paquete `java.awt.event`.

`java.awt.event` siguen utilizándose con los componentes Swing. También se han agregado tipos de eventos adicionales, específicos para varios tipos de componentes Swing. Los nuevos tipos de eventos de los componentes Swing están definidos en el paquete `javax.swing.event`.

Para procesar un evento de interfaz gráfica de usuario, el programador debe realizar dos tareas clave: registrar un *componente que escucha eventos* e implementar un *manejador de eventos*. Un componente que escucha un evento GUI es un objeto de una clase que implementa una o más de las interfaces que escuchan eventos correspondientes a los paquetes `java.awt.event` y `javax.swing.event`. Muchos de los tipos de componentes que escuchan eventos son comunes para los componentes Swing y AWT. Dichos tipos se definen en el paquete `java.awt.event` y muchos de ellos aparecen en la figura 29.6 [Nota: Un fondo sombreado indica una interfaz en el diagrama]. Los tipos adicionales de componentes que escuchan eventos, específicos de los componentes Swing, se definen en el paquete `javax.swing.event`.

Un objeto componente “escucha” tipos específicos de eventos generados en el mismo objeto, o generados por otros objetos (por lo general componentes GUI) en un programa. Un manejador de eventos es un método que se invoca automáticamente en respuesta a un tipo específico de evento. Cada interfaz que escucha eventos especifica uno o más métodos manejadores de eventos que deben definirse en la clase que implementa a la interfaz. Recuerde que las interfaces definen métodos **abstract**. Cualquier clase que implemente a una interfaz deberá definir todos los métodos de esa interfaz; en caso contrario, será una clase **abstract** y no podrá utilizarse para crear objetos. Al uso de componentes que escuchan eventos en el manejo de eventos se conoce como *modelo de delegación de eventos*; el procesamiento de un evento se delega a un objeto específico en el programa.

Cuando ocurre un evento, el componente GUI que interactuó con el usuario notifica a sus componentes de escucha registrados por medio de una llamada al método manejador de eventos apropiado de cada componente de escucha. Por ejemplo, cuando el usuario oprime la tecla *Entrar* en un objeto `JTextField`, se hace una llamada al método `actionPerformed` del componente de escucha registrado. ¿Cómo se registró el manejador de eventos? ¿Cómo es que el componente GUI sabe llamar a `actionPerformed` y no a otro método manejador de eventos? Como parte del siguiente ejemplo, responderemos a esas preguntas y mostraremos un diagrama de la interacción.

29.5 JTextField y JPasswordField

Los objetos **JTextField** y **JPasswordField** (del paquete **javax.swing**) son áreas de una sola línea en las que el usuario puede introducir texto a través del teclado, o simplemente puede mostrarse texto. Un objeto **JPasswordField** muestra que se escribe un carácter a medida que el usuario introduce los caracteres, pero los oculta debido a que supone que representan una contraseña que sólo debe conocer el usuario. Cuando el usuario escribe datos en un objeto **JTextField** o **JPasswordField** y oprime la tecla *Entrar*, se produce un evento de acción. Si un componente que escucha eventos está registrado, el evento se procesa y los datos del objeto **JTextField** o **JPasswordField** pueden utilizarse en el programa. La clase **JTextField** extiende a la clase **JtextComponent** (del paquete **javax.swing.text**), la cual proporciona muchas características comunes a los componentes Swing basados en texto. La clase **JPasswordField** extiende a **JTextField** y agrega varios métodos específicos para el procesamiento de contraseñas.

Error común de programación 29.3



Utilizar una letra f minúscula en los nombres de las clases JTextField o JPasswordField, es un error de sintaxis.

La aplicación de la figura 29.7 utiliza las clases **JTextField** y **JPasswordField** para crear y manipular cuatro campos. Cuando el usuario oprime *Entrar* en el campo que se encuentra activo en ese momento (el componente activo “tiene la atención”) se despliega un cuadro de diálogo de mensaje, el cual contiene el texto del campo. Cuando ocurre un evento en el objeto **JPasswordField**, la contraseña se revela.

```

1 // Figura 29.7: PruebaCampoTexto.java
2 // Demostración de la clase JTextField.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class PruebaCampoTexto extends JFrame {
8     private JTextField texto1, texto2, texto3;
9     private JPasswordField contrasenia;
10
11    public PruebaCampoTexto()
12    {
13        super( "Prueba de JTextField y JPasswordField" );
14
15        Container c = getContentPane();
16        c.setLayout( new FlowLayout() );
17
18        // crea el campo de texto con tamaño predeterminado
19        texto1 = new JTextField( 10 );
20        c.add( texto1 );
21
22        // crea el campo de texto con texto predeterminado
23        texto2 = new JTextField( "Escriba el texto aqui" );
24        c.add( texto2 );
25
26        // crea el campo de texto con texto predeterminado, con
27        // 20 elementos visibles y sin manejador de eventos
28        texto3 = new JTextField( "Campo de texto no editable", 20 );
29        texto3.setEditable( false );
30        c.add( texto3 );
31

```

Figura 29.7 Demostración de **JTextField** y **JPasswordField**; **PruebaCampoTexto.java**.
(Parte 1 de 3.)

```
32      // crea el campo de contraseña con texto predeterminado
33      contrasenia = new JPasswordField( "Texto oculto" );
34      c.add( contrasenia );
35
36      ManejadorCampoTexto manejador = new ManejadorCampoTexto();
37      texto1.addActionListener( manejador );
38      texto2.addActionListener( manejador );
39      texto3.addActionListener( manejador );
40      contrasenia.addActionListener( manejador );
41
42      setSize( 325, 100 );
43      show();
44 } // fin del constructor de PruebaCampoTexto
45
46 public static void main( String args[] )
47 {
48     PruebaCampoTexto ap = new PruebaCampoTexto();
49
50     ap.addWindowListener(
51         new WindowAdapter() {
52             public void windowClosing( WindowEvent e )
53             {
54                 System.exit( 0 );
55             } // fin del método windowClosing
56         } // fin de la clase interna anónima
57     ); // fin de addWindowListener
58 } // fin de main
59
60 // clase interna privada para el manejo de eventos
61 private class ManejadorCampoTexto implements ActionListener {
62     public void actionPerformed( ActionEvent evento )
63     {
64         String cadena = "";
65
66         if ( evento.getSource() == texto1 )
67             cadena = "texto1: " + evento.getActionCommand();
68         else if ( evento.getSource() == texto2 )
69             cadena = "texto2: " + evento.getActionCommand();
70         else if ( evento.getSource() == texto3 )
71             cadena = "texto3: " + evento.getActionCommand();
72         else if ( evento.getSource() == contrasenia ) {
73             JPasswordField contra =
74                 (JPasswordField) evento.getSource();
75             cadena = "contraseña: " +
76                 new String( contra.getPassword() );
77         } // fin de else if
78
79         JOptionPane.showMessageDialog( null, cadena );
80     } // fin del método actionPerformed
81 } // fin de la clase ManejadorCampoTexto
82 } // fin de la clase PruebaCampoTexto
```

Figura 29.7 Demostración de **JTextField** y **JPasswordField**: **PruebaCampoTexto.java**.
(Parte 2 de 3.)

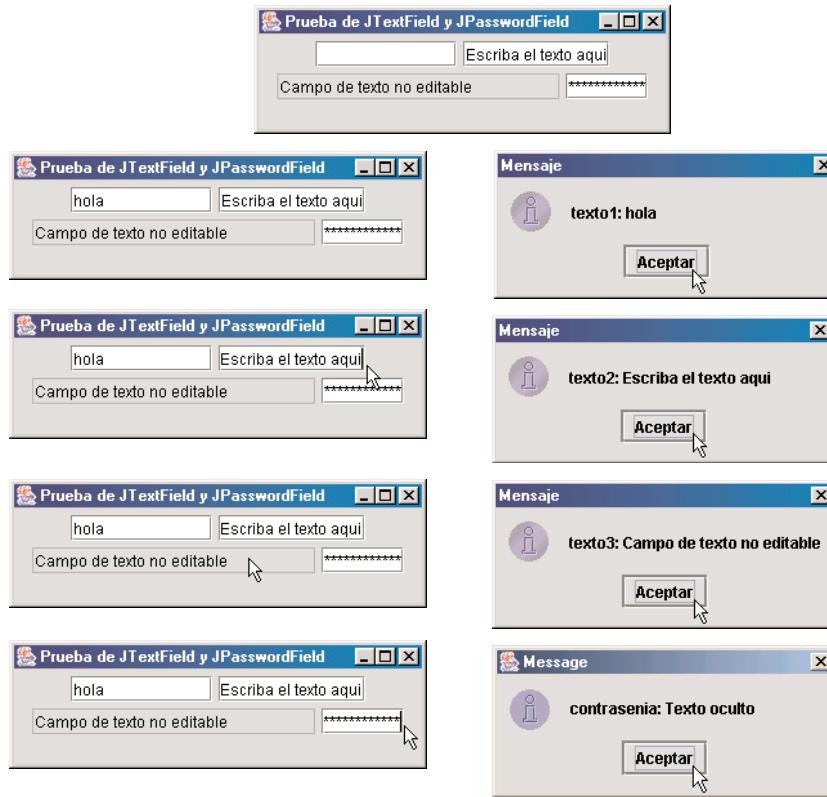


Figura 29.7 Demostración de **JTextField** y **JPasswordField**; **PruebaCampoTexto.java**. (Parte 3 de 3.)

Las líneas 8 y 9 declaran tres referencias para objetos **JTextField** (**texto1**, **texto2** y **texto3**) y un objeto **JPasswordField** (**contrasenia**). Cada uno de estos objetos son instanciados en el constructor (línea 11). La línea 19

```
texto1 = new JTextField( 10 );
```

define el objeto **texto1** de **JTextField** con 10 columnas de texto. El ancho del campo de texto será el ancho en pixeles del carácter promedio en el tipo de letra actual del campo de texto, multiplicado por 10. La línea 20 agrega **texto1** al panel de contenido.

La línea 23

```
texto2 = new JTextField( "Escriba el texto aqui" );
```

define el objeto **texto2** de **JTextField** con el texto inicial “**Escriba el texto aqui**” que muestra el campo de texto. El ancho del campo de texto se determina de acuerdo con el texto. La línea 24 agrega **texto2** al panel de contenido.

La línea 28

```
texto3 = new JTextField( "Campo de texto no editable", 20 );
```

define el objeto **texto3** de **JTextField** y hace una llamada al constructor de **JTextField** con dos argumentos: el texto predeterminado “**Campo de texto no editable**” que se muestra en el campo de texto y el número de columnas (20). El ancho del campo de texto se determina de acuerdo con el número de columnas especificadas. La línea 29

```
texto3.setEditable( false );
```

utiliza el método **`setEditable`** (heredado en **`JTextField`** desde la clase **`JTextComponent`**) para indicar que el usuario no puede modificar el texto del campo de texto. La línea 30 agrega **`texto3`** al panel de contenido.

La línea 33

```
contrasenia = new JPasswordField( "Texto oculto" );
```

define el objeto **`contrasenia`** de **`JPasswordField`** con el texto **"Texto oculto"** a desplegarse en el campo de texto. El ancho del campo de texto se determina de acuerdo con el texto. Observe que el texto aparece como una cadena de asteriscos, cuando se ejecuta el programa. La línea 34 agrega **`contrasenia`** al panel de contenido.

Para el manejo de eventos de este ejemplo, definimos la clase interna **`ManejadorCampoTexto`** (líneas 61 a 81). El manejador de la clase **`JTextField`** (que en breve describiremos detalladamente) implementa la interfaz **`ActionListener`**. Por lo tanto, toda instancia de la clase **`ManejadorCampoTexto`** es un **`ActionListener`**. La línea 36

```
ManejadorCampoTexto manejador = new ManejadorCampoTexto();
```

define una instancia de la clase **`ManejadorCampoTexto`** y la asigna a la referencia **`manejador`**. Esta instancia se utilizará como el objeto componente que escucha eventos para los objetos **`JTextField`** y para el objeto **`JPasswordField`** de este ejemplo.

Las líneas 37 a 40

```
texto1.addActionListener( manejador );
texto2.addActionListener( manejador );
texto3.addActionListener( manejador );
contrasenia.addActionListener( manejador );
```

son las instrucciones de registro de eventos que especifican el objeto componente que escucha eventos para cada uno de los tres objetos **`JTextField`** y para el objeto **`JPasswordField`**. Al ejecutarse estas instrucciones, el objeto al que **`manejador`** hace referencia se queda *escuchando eventos* (es decir, se le notificará cuando ocurra un evento) en estos cuatro objetos. En cada caso, se hace una llamada al método **`addActionListener`** de la clase **`JTextField`** para registrar el evento. El método **`addActionListener`** recibe como su argumento un objeto **`ActionListener`**. Por lo tanto, podrá proporcionarse cualquier objeto de una clase que implemente la interfaz **`ActionListener`** (es decir, cualquier objeto que sea un **`ActionListener`**) como argumento de este método. El objeto al que **`manejador`** hace referencia es un **`ActionListener`**, ya que su clase implementa la interfaz **`ActionListener`**. Ahora, cuando el usuario oprime *Entrar* en cualquiera de estos cuatro campos, se hace una llamada al método **`actionPerformed`** (línea 62) de la clase **`ManejadorCampoTexto`** para manejar el evento.

Observación de ingeniería de software 29.2



El componente que escucha un evento dado deberá implementar la interfaz para escuchar eventos apropiada.

El método **`actionPerformed`** utiliza el método **`getSource`** de su argumento **`ActionEvent`** para determinar el componente GUI con el que interactuó el usuario, y crea un objeto **`String`** para mostrarlo en un cuadro de diálogo de mensajes. El método **`getActionCommand`** de **`ActionEvent`** devuelve el texto en el objeto **`JTextField`** que generó el evento. Si el usuario interactuó con el objeto **`JPasswordField`**, las líneas 73 y 74

```
JPasswordField contra =
(JPasswordField) evento.getSource();
```

convierten la referencia **`Component`** devuelta por **`evento.getSource()`** en una referencia **`JPasswordField`**, de manera que las líneas 75 y 76

```
cadena = "contrasenia: " +
new String( contra.getPassword() );
```

puedan utilizar el método **`getPassword`** de **`JPasswordField`** para obtener la contraseña y crear el objeto **`String`** a desplegar en pantalla. El método **`getPassword`** devuelve la contraseña como un arreglo de tipo

char que se utiliza como argumento para un constructor **String**, para crear un objeto **String**. La línea 79 muestra un cuadro de mensaje que indica el nombre de la referencia del componente GUI y el texto que escribió el usuario en el campo.

Observe que hasta un objeto **JTextField** no editable puede generar un evento. También observe que el texto real de la contraseña aparece cuando oprime *Entrar* en el objeto **JPasswordField** (por supuesto, ¡normalmente no haría esto!).

Error común de programación 29.4



Olvidar registrar un objeto manejador de eventos para un tipo de evento de un componente GUI en particular, da como resultado que no se manejen los eventos de ese componente.

Utilizar una clase separada para definir un componente que escucha eventos es una práctica común de programación para separar la interfaz GUI de la implementación de su manejador de eventos. En el resto de este capítulo, muchos programas utilizan clases separadas de componentes que escuchan eventos para procesar eventos GUI, en un intento por hacer que el código sea más reutilizable. Cualquier clase con el potencial para reutilizarse más allá del ejemplo en el que se introdujo se ha colocado en un paquete, de manera que puede importarse desde otros programas para su reutilización.

Buena práctica de programación 29.5



Utilice clases separadas para procesar eventos GUI.

Observación de ingeniería de software 29.3



Utilizar clases separadas para manejar eventos GUI produce componentes de software más reutilizables, confiables y legibles, los cuales pueden colocarse en paquetes y utilizarse en muchos programas.

29.5.1 Cómo funciona el manejo de eventos

Ahora veamos cómo funciona el mecanismo de manejo de eventos, utilizando el objeto `textol` de `JTextField` del ejemplo anterior. Tenemos dos preguntas de la sección 29.4 que no hemos contestado:

1. ¿Cómo quedó registrado el manejador de eventos?
 2. ¿Cómo es que el componente GUI sabe llamar a **actionPerformed** y no a otro método manejador de eventos?

Respondemos a la primera pregunta por medio del proceso de registro de eventos que se lleva a cabo en las líneas 37 a 40 del programa. La figura 29.8 muestra un diagrama del objeto `textol` de `JTextField` y su manejador de eventos registrado.

Todo objeto **JComponent** tiene un objeto de la clase **EventListenerList** (del paquete **javax.swing.event**) llamado *listenerList*, como una variable de instancia. Todos los componentes de es-

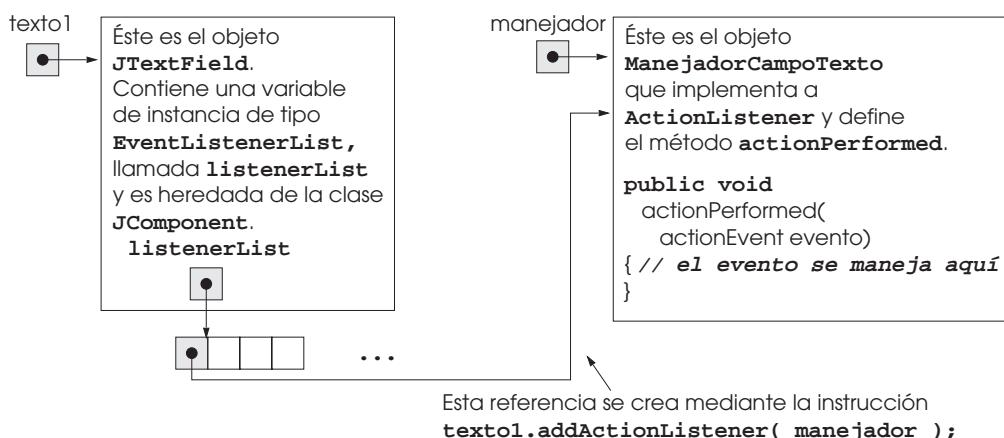


Figura 29.8 Registro de eventos para el objeto `texto1` de `JTextField`.

cucha registrados se almacenan en el objeto **listenerList** (diagramado como un arreglo en la figura 29.8). Cuando se ejecuta la instrucción

```
texto1.addActionListener( manejador );
```

de la figura 29.7, se coloca una nueva entrada en el objeto **listenerList** para el objeto **texto1** de **JTextField**, la cual indica la referencia al objeto de escucha y el tipo de componente de escucha (en este caso, **ActionListener**).

El tipo es importante para responder a la segunda pregunta: ¿cómo es que el componente GUI sabe que debe llamar a **actionPerformed**, en vez de a cualquier otro método manejador de eventos? En realidad, todo objeto **JComponent** soporta varios tipos de eventos distintos, incluyendo *eventos del ratón*, *eventos de teclas* y otros más. Cuando ocurre un evento, éste se *despacha* (se envía) solamente a los componentes apropiados que escuchan eventos. El proceso de despachar un evento consiste simplemente en llamar al método manejador de eventos para cada componente de escucha registrado para ese tipo de evento.

Cada tipo de evento tiene su correspondiente interfaz para escuchar eventos. Por ejemplo, los eventos **ActionEvent** son manejados por objetos **ActionListener**, los eventos **MouseEvent** son manejados por objetos **MouseListener** (y por objetos **MouseMotionListener**, como veremos más adelante) y los eventos **KeyEvent** son manejados por objetos **KeyListener**. Cuando se genera un evento debido a la interacción del usuario con un componente, éste recibe un *número de identificación (ID)* de evento único, el cual especifica el tipo de evento que ocurrió. El componente GUI utiliza el ID de evento para decidir el tipo de componente de escucha al que debe enviarse el evento, junto con el método al que debe llamarse. En el caso de un evento **ActionEvent**, éste se envía a cada método registrado **actionPerformed** de **ActionListener** (el único método de la interfaz **ActionListener**). En el caso de un **MouseEvent**, éste se envía a todos los objetos **MouseListener** (o **MouseMotionListener**) registrados. El ID del evento **MouseEvent** determina a cuál de los siete distintos métodos manejadores de eventos de ratón se llama. Los componentes GUI manejan toda esta lógica de decisión por usted. Explicaremos otros tipos de eventos e interfaces para escuchar eventos conforme las necesitemos, cuando analicemos cada nuevo componente.

29.6 JTextArea

Los objetos **JTextArea** proporcionan un área para manipular varias líneas de texto. Al igual que la clase **JTextField**, la clase **JTextArea** hereda de **JTextComponent**, la cual define métodos comunes para objetos **JTextField**, **JTextarea** y varios otros componentes GUI basados en texto.

La aplicación de la figura 29.9 muestra el uso de objetos **JTextArea**. Un objeto **JTextArea** muestra texto que el usuario puede seleccionar. El segundo objeto **JTextArea** no puede editarse, y su propósito es mostrar el texto que el usuario seleccionó en el primer objeto **JTextArea**. Los objetos **JTextArea** no tienen eventos de acción como los objetos **JTextField**. A menudo, un *evento externo* (es decir, un evento generado por otro componente GUI) indica cuándo debe procesarse el texto de un objeto **JTextArea**. Por ejemplo, para enviar un mensaje de correo electrónico, el usuario generalmente hace clic en un botón **Enviar** para tomar el texto del mensaje y enviarlo al destinatario. De manera similar, cuando se edita un documento en un procesador de palabras, usted por lo general guarda el archivo seleccionando un elemento de menú llamado **Guardar** o **Guardar como....** En este programa, el botón **Copiar >>>** genera el evento externo que hace que el texto seleccionado del objeto **JTextArea** de la izquierda se copie y se muestre en el objeto **JTextArea** de la derecha.

Observación de apariencia visual 29.4



A menudo, un evento externo determina cuándo debe procesarse el texto de un objeto **JTextArea**.

```
1 // Figura 29.9: DemoAreaTexto.java
2 // Cómo copiar texto seleccionado de un área de texto hacia otra.
3 import java.awt.*;
```

Figura 29.9 Cómo copiar el texto seleccionado de un área de texto a otra; **DemoAreaTexto.java**. (Parte 1 de 3.)

```
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class DemoAreaTexto extends JFrame {
8     private JTextArea t1, t2;
9     private JButton copiar;
10
11    public DemoAreaTexto()
12    {
13        super( "Demostracion de TextArea" );
14
15        Box b = Box.createHorizontalBox();
16
17        String cadena = "Esta es una cadena de demostracion para\n" +
18                    "mostrar como copiar texto\n" +
19                    "de un objeto TextArea a \n" +
20                    "otro objeto TextArea, usando un\n" +
21                    "evento externo\n";
22
23        t1 = new JTextArea( cadena, 10, 15 );
24        b.add( new JScrollPane( t1 ) );
25
26        copiar = new JButton( "Copiar >>>" );
27        copiar.addActionListener(
28            new ActionListener() {
29                public void actionPerformed( ActionEvent e )
30                {
31                    t2.setText( t1.getSelectedText() );
32                } // fin del método actionPerformed
33            } // fin de la clase interna anónima
34        ); // fin de addActionListener
35        b.add( copiar );
36
37        t2 = new JTextArea( 10, 15 );
38        t2.setEditable( false );
39        b.add( new JScrollPane( t2 ) );
40
41        Container c = getContentPane();
42        c.add( b );
43        setSize( 425, 200 );
44        show();
45    } // fin del constructor de DemoAreaTexto
46
47    public static void main( String args[] )
48    {
49        DemoAreaTexto ap = new DemoAreaTexto();
50
51        ap.addWindowListener(
52            new WindowAdapter() {
53                public void windowClosing( WindowEvent e )
54                {
55                    System.exit( 0 );
56                } // fin del método windowClosing
57            } // fin de la clase interna anónima
```

Figura 29.9 Cómo copiar el texto seleccionado de un área de texto a otra; **DemoAreaTexto.java**.
(Parte 2 de 3.)

```

58      ); // fin de addWindowListener
59  } // fin de main
60 } // fin de la clase DemoTextArea

```

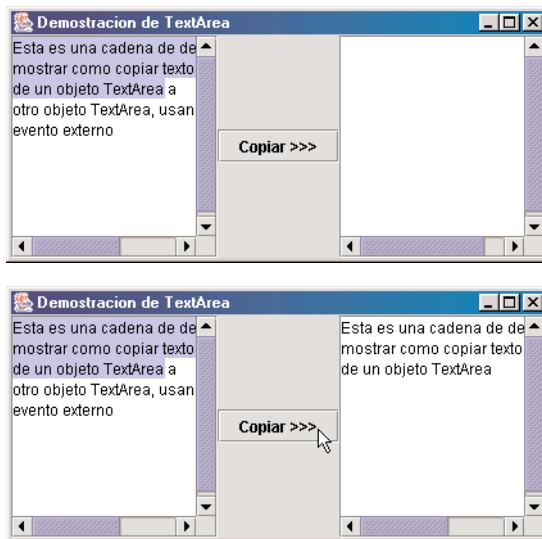


Figura 29.9 Cómo copiar el texto seleccionado de un área de texto a otra; **DemoTextArea.java**. (Parte 3 de 3.)

En el método constructor, la línea 15

```
Box b = Box.createHorizontalBox();
```

crea un *contenedor Box* (del paquete **javax.swing**) al que se agregarán los componentes GUI. La clase **Box** es una subclase de **java.awt.Container** que utiliza un administrador de diseño **BoxLayout** para ordenar los componentes GUI, ya sea en forma horizontal o vertical. En la sección 29.11, hablaremos más sobre los administradores de diseño. La clase **Box** proporciona el método estático **createHorizontalBox** para crear un objeto **Box** que ordene automáticamente de izquierda a derecha los componentes que se le agreguen, conforme se vayan agregando.

La aplicación crea instancias de objetos **JTextArea** y los asigna a las referencias **t1** (línea 23) y **t2** (línea 37). Cada objeto **JTextArea** tiene 10 filas y 15 columnas visibles. La línea 23

```
t1 = new JTextArea( cadena, 10, 15 );
```

especifica que la cadena predeterminada **cadena** debe mostrarse en el objeto **JTextArea**. Un objeto **JTextArea** no proporciona barras de desplazamiento, en caso de que haya más texto del que pueda mostrarse en ese objeto. Por esta razón, la línea 24

```
b.add( new JScrollPane( t1 ) );
```

crea un objeto **JScrollPane**, el cual se inicializa con el objeto **t1** de **JTextArea** y con desplazamiento horizontal y vertical, según sea necesario. Después, el objeto **JScrollPane** se agrega directamente al contenedor **Box** llamado **b**.

Las líneas 26 a 35 crean una instancia de un objeto **JButton** y la asignan a la referencia **copiar** con la etiqueta “**Copiar >>**”; crean una clase interna anónima para manejar el evento **ActionEvent** de **copiar**; y agregan **copiar** al objeto contenedor **Box** al que **b** hace referencia. Este botón proporciona el evento externo que determina cuándo debe copiarse el texto seleccionado de **t1** a **t2**. Cuando el usuario hace clic en **Copiar >>**, la línea 31

```
t2.setText( t1.getSelectedText() );
```

en `actionPerformed` indica que el método `getSelectedText` (heredado a `JTextArea` desde `JTextComponent`) debe regresar el *texto seleccionado* de `t1`. Para seleccionar el texto, arrastre el ratón sobre éste para resaltarlo. Después, el método `setText` cambia el texto en `t2` por el objeto `String` devuelto.

Las líneas 37 a 39 crean el objeto `t2` de `JTextArea` y lo agregan al contenedor `b` de `Box`. Las líneas 41 y 42 obtienen el panel de contenido para la ventana y agregan el objeto `Box` al panel de contenido. El diseño del panel de contenido es administrado por un objeto `BorderLayout`, el cual describiremos en la sección 29.11.2.

[Nota: Cuando el texto llega al lado derecho de un objeto `JTextArea`, algunas veces es conveniente que el resto del texto pase a la siguiente línea. A esto se le conoce como *envoltura automática de palabras*.]



Observación de apariencia visual 29.5

Para proporcionar la funcionalidad de envoltura automática de palabras para un objeto `JTextArea`, invoque al método `setLineWrap` con un argumento `true`.

[Nota: Usted puede establecer las *directivas de las barras de desplazamiento horizontal y vertical* para el objeto `JScrollPane` al momento de crearlo, o en cualquier momento mediante los métodos `setHorizontalScrollBarPolicy` y `setVerticalScrollBarPolicy` de la clase `JScrollPane`.] La clase `JScrollPane` proporciona las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS  
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que una barra de desplazamiento debe aparecer siempre; las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED  
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que una barra de desplazamiento debe aparecer solamente si es necesario; y las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER  
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que una barra de desplazamiento no debe aparecer nunca. Si la directiva de barra de desplazamiento horizontal se establece como `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, un objeto `JTextArea` adjunto al objeto `JScrollPane` exhibirá un comportamiento de envoltura automática de palabras.

29.7 JButton

Un *botón* es un componente en el que el usuario hace clic para desencadenar una acción específica. Un programa en Java puede utilizar varios tipos de botones, que incluyen *botones de comando*, *casillas de verificación*, *botones de commutación* y *botones de opción*. La figura 29.10 muestra la jerarquía de herencia de los botones Swing que veremos en este capítulo. Como puede ver en el diagrama, todos los tipos de botones son subclases de `AbstractButton` (del paquete `javax.swing`), el cual define muchas de las características comunes para los botones Swing. En esta sección nos concentraremos en los botones que se utilizan generalmente para iniciar un comando. En las siguientes secciones veremos otros tipos de botones.

Un botón de comando genera un evento `ActionEvent` cuando el usuario hace clic con el ratón sobre el botón. Los botones de comando se crean con la clase `Jbutton`, la cual hereda de la clase `AbstractBut-`

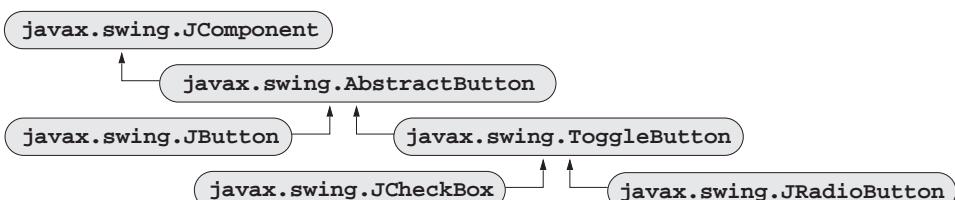


Figura 29.10 La jerarquía de botones.

ton. Al texto en la cara de un objeto **JButton** se le llama *etiqueta del botón*. Una GUI puede tener muchos objetos **JButton**, pero cada etiqueta de botón debe ser única.

Observación de apariencia visual 29.6

 Tener más de un objeto **JButton** con la misma etiqueta hace que los objetos **JButton** sean ambiguos para el usuario. Asegúrese de proporcionar una etiqueta única para cada botón.

La aplicación de la figura 29.11 crea dos objetos **JButton** y muestra que los objetos **JButton** (como los objetos **JLabel**) soportan el despliegue de objetos **Icon**. El manejo de eventos para los botones se lleva a cabo mediante una sola instancia de la clase interna **ManejadorBoton** (línea 52).

La línea 8 declara dos referencias a instancias de la clase **JButton**: **botonSimple** y **botonElegante**, las cuales se instancian en el constructor (línea 10).

La línea 18

```
botonSimple = new JButton( "Boton simple" );
```

crea **botonSimple** con la etiqueta de botón **"Boton simple"**. La línea 19 agrega el botón al panel de contenido.

Un objeto **JButton** puede mostrar objetos **Icon**. Para proporcionar al usuario un nivel adicional de interactividad visual con la GUI, un objeto **JButton** puede tener también un objeto **Icon de sustitución**: un objeto **Icon** que aparece cuando el ratón se posiciona sobre el botón. El ícono en el botón cambia a medida que el ratón se aleja y se acerca al área del botón en la pantalla. Las líneas 21 y 22:

```
Icon insecto1 = new ImageIcon( "insecto1.gif" );
Icon insecto2 = new ImageIcon( "insecto2.gif" );
```

crean dos objetos **ImageIcon** que representan al objeto **Icon** predeterminado y al objeto **Icon** de sustitución para el objeto **JButton** creado en la línea 23. Ambas instrucciones asumen que los archivos de imagen están almacenados en el mismo directorio que el programa (por lo general, éste es el caso para las aplicaciones que utilizan imágenes).

```

1 // Figura 29.11: PruebaBoton.java
2 // Creación de objetos JButton.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class PruebaBoton extends JFrame {
8     private JButton botonSimple, botonElegante;
9
10    public PruebaBoton()
11    {
12        super( "Prueba de botones" );
13
14        Container c = getContentPane();
15        c.setLayout( new FlowLayout() );
16
17        // crea los botones
18        botonSimple = new JButton( "Boton simple" );
19        c.add( botonSimple );
20
21        Icon insecto1 = new ImageIcon( "insecto1.gif" );
22        Icon insecto2 = new ImageIcon( "insecto2.gif" );
23        botonElegante = new JButton( "Boton elegante", insecto1 );
24        botonElegante.setRolloverIcon( insecto2 );

```

Figura 29.11 Demostración de botones de comando y de eventos de acción; **PruebaBoton.java**.
(Parte 1 de 2.)

```

25         c.add( botonElegante );
26
27         // crea una instancia de la clase interna ManejadorBoton
28         // para usarla en el manejo de eventos de botón
29         ManejadorBoton manejador = new ManejadorBoton();
30         botonElegante.addActionListener( manejador );
31         botonSimple.addActionListener( manejador );
32
33         setSize( 300, 100 );
34         show();
35     } // fin del constructor de PruebaBoton
36
37     public static void main( String args[] )
38     {
39         PruebaBoton ap = new PruebaBoton();
40
41         ap.addWindowListener(
42             new WindowAdapter() {
43                 public void windowClosing( WindowEvent e )
44                 {
45                     System.exit( 0 );
46                 } // fin del método windowClosing
47             } // fin de la clase interna anónima
48         ); // fin de addWindowListener
49     } // fin de main
50
51     // clase interna para el manejo de eventos de botón
52     private class ManejadorBoton implements ActionListener {
53         public void actionPerformed( ActionEvent e )
54         {
55             JOptionPane.showMessageDialog( null,
56                 "Usted oprimio: " + e.getActionCommand() );
57         } // fin del método actionPerformed
58     } // fin de la clase ManejadorBoton
59 } // fin de la clase PruebaBoton

```

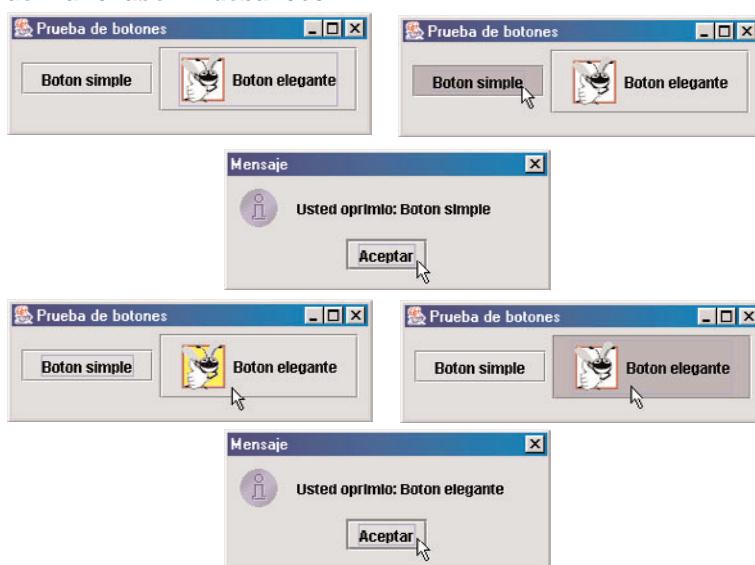


Figura 29.11 Demostración de botones de comando y de eventos de acción; **PruebaBoton.java**. (Parte 2 de 2.)

La línea 23

```
botonElegante = new JButton( "Botón elegante", insecto1 );
```

crea el objeto **botonElegante** con el texto predeterminado **"Botón elegante"** y el objeto **insecto1** de **Icon**. De manera predeterminada, el texto se despliega a la derecha del ícono. La línea 24

```
botonElegante.setRollOverIcon( insecto2 );
```

utiliza el método **setRollOverIcon** (la clase **JButton** lo hereda de la clase **AbstractButton**) para especificar la imagen que se despliega en el botón cuando el usuario coloca el ratón sobre él. La línea 25 agrega el botón al panel de contenido.

Observación de apariencia visual 29.7



El uso de iconos de sustitución para objetos **JButton** proporciona al usuario una retroalimentación visual, la cual le indica que, si hace clic en el ratón, se realizará la acción del botón.

Los objetos **JButton** (como los **JTextFields**) generan **ActionEvents**. Como mencionamos anteriormente, un **ActionEvent** puede ser procesado por cualquier objeto **ActionListener**. Las líneas 29 a 31

```
ManejadorBoton manejador = new ManejadorBoton();
botonElegante.addActionListener( manejador );
botonSimple.addActionListener( manejador );
```

registran un objeto **ActionListener** para cada objeto **JButton**. La clase interna **ManejadorBoton** (líneas 52 a 58) define el método **actionPerformed** para mostrar un cuadro de diálogo de mensaje que contiene la etiqueta del botón que el usuario oprimió. El método **getActionCommand** de **ActionEvent** devuelve la etiqueta del botón que generó el evento.

29.8 JCheckBox

Los componentes GUI de Swing contienen tres tipos de *botones de estado*: **JToggleButton**, **JCheckBox** y **JRadioButton**, los cuales tienen valores de tipo encendido/apagado o verdadero/falso. Los **JToggleButtons** se utilizan frecuentemente con las *barras de herramientas* (conjuntos de pequeños botones que se encuentran generalmente en una barra, en la parte superior de una ventana). Las clases **JCheckBox** y **JRadioButton** son subclases de **JToggleButton**. Un **JRadioButton** es distinto de un **JCheckBox** en cuanto a que generalmente hay varios objetos **JRadioButton** agrupados, y sólo puede seleccionarse uno de los objetos **JRadioButton** (como **true**) en cualquier momento dado. En esta sección explicaremos la clase **JCheckBox**.

Observación de apariencia visual 29.8



La clase **AbstractButton** soporta que se despliegue texto e imágenes en un botón, por lo que todas las subclases de **AbstractButton** también soportan el despliegue de texto e imágenes.

La aplicación de la figura 29.12 utiliza dos objetos **JCheckBox** para cambiar el estilo de la fuente del texto desplegado en un objeto **JTextField**. Un objeto **JCheckBox** aplica un estilo de negritas al seleccionarlo, y el otro aplica un estilo de cursivas al seleccionarlo. Si se seleccionan ambos, el estilo de la fuente será en negritas y cursivas. Cuando el programa se ejecuta por primera vez, ninguno de los objetos **JCheckBox** está seleccionado (**true**).

```
1 // Figura 29.12: PruebaCasillaVerificacion.java
2 // Creación de botones JCheckBox.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
```

Figura 29.12 Programa que crea dos botones **JCheckBox**; **PruebaCasillaVerificacion.java**.
(Parte 1 de 3.)

```
7  public class PruebaCasillaVerificacion extends JFrame {
8      private JTextField t;
9      private JCheckBox negrita, cursiva;
10
11     public PruebaCasillaVerificacion()
12     {
13         super( "Prueba de JCheckBox" );
14
15         Container c = getContentPane();
16         c.setLayout( new FlowLayout() );
17
18         t = new JTextField( "Observe como cambia el estilo de la fuente", 28 );
19         t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
20         c.add( t );
21
22         // crea los objetos casilla de verificación
23         negrita = new JCheckBox( "Negrita" );
24         c.add( negrita );
25
26         cursiva = new JCheckBox( "Cursiva" );
27         c.add( cursiva );
28
29         ManejadorCasillaVerificacion manejador =
30             new ManejadorCasillaVerificacion();
31         negrita.addItemListener( manejador );
32         cursiva.addItemListener( manejador );
33
34         addWindowListener(
35             new WindowAdapter() {
36                 public void windowClosing( WindowEvent e )
37                 {
38                     System.exit( 0 );
39                 } // fin del método windowClosing
40             } // fin de la clase interna anónima
41         ); // fin de addWindowListener
42
43         setSize( 325, 100 );
44         show();
45     } // fin del constructor de PruebaCasillaVerificacion
46
47     public static void main( String args[] )
48     {
49         new PruebaCasillaVerificacion();
50     }
51
52     private class ManejadorCasillaVerificacion implements ItemListener {
53         private int valNegrita = Font.PLAIN;
54         private int valCursiva = Font.PLAIN;
55
56         public void itemStateChanged( ItemEvent e )
57         {
58             if ( e.getSource() == negrita )
59                 if ( e.getStateChange() == ItemEvent.SELECTED )
60                     valNegrita = Font.BOLD;
```

Figura 29.12 Programa que crea dos botones **JCheckBox**: **PruebaCasillaVerificacion.java**. (Parte 2 de 3.)

```

60         else
61             valNegrita = Font.PLAIN;
62
63         if ( e.getSource() == cursiva )
64             if ( e.getStateChange() == ItemEvent.SELECTED )
65                 valCursiva = Font.ITALIC;
66             else
67                 valCursiva = Font.PLAIN;
68
69         t.setFont(
70             new Font( "TimesRoman", valNegrita + valCursiva, 14 ) );
71         t.repaint();
72     } // fin del método itemStateChanged
73 } // fin de la clase interna ManejadorCasillaVerificacion
74 } // fin de la clase PruebaCasillaVerificacion

```

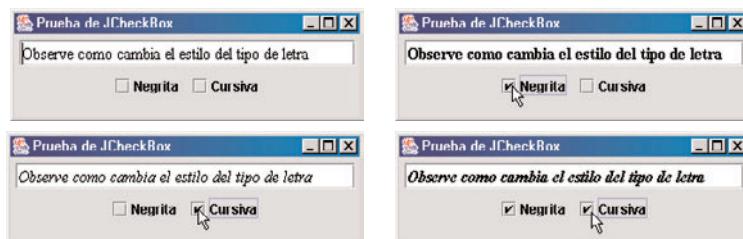


Figura 29.12 Programa que crea dos botones **JCheckBox**; **PruebaCasillaVerificacion.java**. (Parte 3 de 3.)

Una vez que se crea y se inicializa el objeto **JTextField**, la línea 19

```
t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
```

establece en **TimesRoman**, **PLAIN** y en 14 puntos a la fuente del objeto **JTextField**. Después, el constructor crea dos objetos **JCheckBox** mediante las líneas 23 y 26

```
negrita = new JCheckBox( "Negrita" );
cursiva = new JCheckBox( "Cursiva" );
```

El objeto **String** que se pasa al constructor es la *etiqueta de la casilla de verificación* que aparece a la derecha del objeto **JCheckBox**, de manera predeterminada.

Cuando el usuario hace clic en un objeto **JCheckBox** se genera un **ItemEvent**, el cual puede ser manejado por un **ItemListener** (cualquier objeto de una clase que implemente la interfaz **ItemListener**). Un objeto **ItemListener** debe definir el método **itemStateChanged**. En este ejemplo, el manejo de eventos se lleva a cabo mediante una instancia de la clase interna **ManejadorCasillaVerificacion** (líneas 51 a 73). Las líneas 29 a 31

```
ManejadorCasillaVerificacion manejador = new ManejadorCasillaVerificacion();
negrita.addItemListener( manejador );
cursiva.addItemListener( manejador );
```

crean una instancia de la clase **ManejadorCasillaVerificacion** y se registra con el método **addItemListener** como el objeto **ItemListener** para los objetos **JCheckBox negrita** y **cursiva**.

Cuando el usuario hace clic en cualquiera de los objetos **JCheckBox**, **negrita** o **cursiva**, se llama al método **itemStateChanged** (línea 55). Este método utiliza a **e.getSource()** para determinar en cuál objeto **JCheckBox** se hizo clic. Si fue en el objeto **negrita** de **JCheckBox**, la estructura **if/else** de las líneas 58 a 61

```
if ( e.getStateChange() == ItemEvent.SELECTED )
    valNegrita = Font.BOLD;
else
    valNegrita = Font.PLAIN;
```

utiliza el método `getStateChange` de `ItemEvent` para determinar el estado del botón (`ItemEvent.SELECTED` o `ItemEvent.DESELECTED`). Si se selecciona el estado `negrita`, a la variable entera `valNegrita` se le asigna `Font.BOLD`; de lo contrario, a `valNegrita` se le asigna `Font.PLAIN`. Si hace clic en el objeto `cursiva` de `JCheckBox`, se ejecuta una estructura `if/else` similar. Si se selecciona el estado `cursiva`, a la variable entera `valCursiva` se le asigna `Font.ITALIC`; de lo contrario, a `valCursiva` se le asigna `Font.PLAIN`. La suma de `valNegrita` y `valCursiva` se utiliza en las líneas 69 y 70 como el estilo del nuevo tipo de fuente para el objeto `JTextField`.

29.9 JComboBox

Un *cuadro combinado* (algunas veces conocido como *lista desplegable*) proporciona una lista de elementos, de los cuales el usuario puede seleccionar uno. Los cuadros combinados se implementan mediante la clase `JComboBox`, la cual hereda de la clase `JComponent`. Los objetos `JComboBox` generan eventos `ItemEvent`, al igual que los objetos `JCheckBox` y `JRadioButton`.

La aplicación de la figura 29.13 utiliza un objeto `JComboBox` para proporcionar una lista de cuatro nombres de archivos de imágenes. Al seleccionar un archivo de imagen, la imagen correspondiente aparece como un ícono en un objeto `JLabel`. En las capturas de las imágenes de este programa aparece la lista `JComboBox` después de haber hecho la selección, para ilustrar cuál nombre de archivo de imagen se seleccionó.

```

1 // Figura 29.13: PruebaCuadroCombinado.java
2 // Uso de un objeto JComboBox para seleccionar una imagen a desplegar.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class PruebaCuadroCombinado extends JFrame {
8     private JComboBox imagenes;
9     private JLabel etiqueta;
10    private String nombres[] =
11        { "insecto1.gif", "insecto2.gif",
12          "insectoviaje.gif", "insectananim.gif" };
13    private Icon iconos[] =
14        { new ImageIcon( nombres[ 0 ] ),
15          new ImageIcon( nombres[ 1 ] ),
16          new ImageIcon( nombres[ 2 ] ),
17          new ImageIcon( nombres[ 3 ] ) };
18
19    public PruebaCuadroCombinado()
20    {
21        super( "Prueba de JComboBox" );
22
23        Container c = getContentPane();
24        c.setLayout( new FlowLayout() );
25
26        imagenes = new JComboBox( nombres );
27        imagenes.setMaximumRowCount( 3 );
28
29        imagenes.addItemListener(
30            new ItemListener() {
31                public void itemStateChanged( ItemEvent e )
32                {
33                    etiqueta.setIcon(
```

Figura 29.13 Programa que utiliza un objeto `JComboBox` para seleccionar un ícono;
`PruebaCuadroCombinado.java`. (Parte 1 de 2.)

```

34         iconos[ imagenes.getSelectedIndex() ] );
35     } // fin del método itemStateChanged
36   } // fin de la clase interna anónima
37 ); // fin de addItemListener
38
39 c.add( imagenes );
40
41 etiqueta = new JLabel( iconos[ 0 ] );
42 c.add( etiqueta );
43
44 setSize( 350, 100 );
45 show();
46 } // fin del constructor de PruebaCuadroCombinado
47
48 public static void main( String args[] )
49 {
50     PruebaCuadroCombinado ap = new PruebaCuadroCombinado();
51
52     ap.addWindowListener(
53         new WindowAdapter()
54         {
55             public void windowClosing( WindowEvent e )
56             {
57                 System.exit( 0 );
58             } // fin del método windowClosing
59         } // fin de la clase interna anónima
60     ); // fin de addWindowListener
61 } // fin de main
61 } // fin de la clase PruebaCuadroCombinado

```

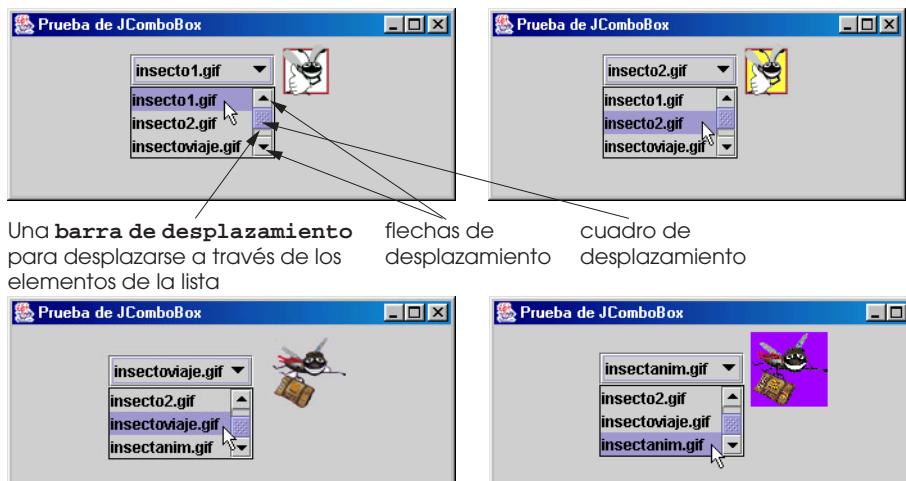


Figura 29.13 Programa que utiliza un objeto **JComboBox** para seleccionar un ícono; **PruebaCuadroCombinado.java**. (Parte 2 de 2.)

Las líneas 13 a 17

```

private Icon iconos[] =
{
    new ImageIcon( nombres[ 0 ] ),
    new ImageIcon( nombres[ 1 ] ),
    new ImageIcon( nombres[ 2 ] ),
    new ImageIcon( nombres[ 3 ] ) };

```

declaran e inicializan el arreglo **iconos** con cuatro nuevos objetos **ImageIcon**. El arreglo **String** llamado **nombres** (definido en las líneas 10 a 12) contiene los nombres de los cuatro archivos de imágenes que están almacenados en el mismo directorio que la aplicación.

La línea 26

```
imagenes = new JComboBox( nombres );
```

crea un **JComboBox**, utilizando los **Strings** del arreglo **nombres** como elementos para la lista. Un *índice* numérico lleva el registro del orden de los elementos del objeto **JComboBox**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento agregado a un objeto **JComboBox** aparece como el elemento actualmente seleccionado, cuando el objeto **JComboBox** aparece en pantalla. Otros elementos se seleccionan haciendo clic en el objeto **JComboBox**. Cuando se hace clic en este objeto, el **JComboBox** se expande en una lista, en la que el usuario puede seleccionar un elemento.

La línea 27

```
imagenes.setMaximumRowCount( 3 );
```

utiliza el método **setMaximumRowCount** de **JComboBox** para establecer el número máximo de elementos que se muestran cuando el usuario hace clic en el objeto **JComboBox**. Si hay más elementos en el objeto **JComboBox** que el número máximo de elementos que aparecen en pantalla, el objeto **JComboBox** proporciona automáticamente una *barra de desplazamiento* (vea la primera imagen capturada en pantalla), la cual permite al usuario ver todos los elementos de la lista. El usuario puede hacer clic en las *flechas de desplazamiento* en la parte superior e inferior de la barra de desplazamiento, para moverse hacia arriba y hacia abajo por la lista, un elemento a la vez, o puede arrastrar el *cuadro de desplazamiento* (que está en medio de la barra de desplazamiento) hacia arriba o hacia abajo para avanzar por la lista. Para arrastrar este cuadro de desplazamiento, mantenga oprimido el botón del ratón con su puntero en el cuadro de desplazamiento, y después mueva el ratón.

Observación de apariencia visual 29.9



Establezca el conteo máximo de filas para un objeto **JComboBox** en un número que evite que la lista se expanda más allá de los límites de la ventana o del applet en que se utilice. Esto garantizará que la lista aparezca correctamente cuando el usuario la expanda.

Las líneas 29 a 37

```
imagenes.addItemListener(
    new ItemListener() {
        public void itemStateChanged( ItemEvent e )
        {
            etiqueta.setIcon(
                iconos[ imagenes.getSelectedIndex() ] );
        } // fin del método itemStateChanged
    } // fin de la clase interna anónima
); // fin de addItemListener
```

registran una instancia de una clase interna anónima que implementa a **ItemListener** como el componente de escucha para el objeto **imagenes** de **JComboBox**. Cuando el usuario hace una selección de **imagenes**, el método **itemStateChanged** (línea 31) establece el objeto **Icon** para **etiqueta**. Para seleccionar el objeto **Icon** del arreglo **iconos**, se determina el número del índice del elemento seleccionado en el objeto **JComboBox** con el método **getSelectedIndex** de la línea 34.

29.10 Manejo de eventos de ratón

En esta sección presentaremos las interfaces que escuchan eventos **MouseListener** y **MouseMotionListener** para manejar *eventos del ratón*. Estos eventos pueden ser atrapados por cualquier componente GUI que se derive de **java.awt.Component**. La figura 29.14 resume los métodos de las interfaces **MouseListener** y **MouseMotionListener**.

Métodos de las interfaces MouseListener y MouseMotionListener

```

public void mousePressed( MouseEvent e )           // MouseListener
    Se llama cuando se oprime un botón del ratón y el puntero de éste se encuentra sobre
    un componente.

public void mouseClicked( MouseEvent e )          // MouseListener
    Se llama cuando se oprime y se suelta un botón del ratón en un componente, sin mover
    el cursor del ratón.

public void mouseReleased( MouseEvent e )         // MouseListener
    Se llama cuando se suelta un botón del ratón, después de oprimirlo. Este evento siempre
    va después de un evento mousePressed.

public void mouseEntered( MouseEvent e )          // MouseListener
    Se llama cuando el cursor del ratón entra a los límites de un componente.

public void mouseExited( MouseEvent e )           // MouseListener
    Se llama cuando el cursor del ratón sale de los límites de un componente.

public void mouseDragged( MouseEvent e )          // MouseMotionListener
    Se llama cuando se oprime el botón del ratón y éste se mueve. Este evento siempre va después
    de una llamada a mousePressed.

public void mouseMoved( MouseEvent e )            // MouseMotionListener
    Se llama cuando el ratón se mueve y el cursor del éste se encuentra sobre un componente.

```

Figura 29.14 Métodos de las interfaces **MouseListener** y **MouseMotionListener**.

Cada uno de los métodos manejadores de eventos de ratón toma un objeto **MouseEvent** como su argumento. Un objeto **MouseEvent** contiene información acerca del evento de ratón que ocurrió, incluso las coordenadas *x* y *y* de la posición en la que ocurrió el evento. Los métodos de **MouseListener** y **MouseMotionListener** se llaman siempre que el ratón interactúa con un objeto **Component**, si hay objetos componentes de escucha registrados para un objeto **Component** específico. El método **mousePressed** se llama cuando se oprime un botón del ratón y el cursor de éste se encuentra sobre un componente. Por medio de los métodos y constantes de la clase **InputEvent** (la superclase de **MouseEvent**), un programa puede determinar cuál fue el botón que oprimió el usuario. El método **mouseClicked** se llama siempre que se suelta un botón del ratón sin moverlo, después de una operación **mousePressed**. El método **mouseReleased** se llama siempre que se suelta un botón del ratón. El método **mouseEntered** se llama cuando el cursor del ratón entra a los límites físicos de un objeto **Component**. El método **mouseExited** se llama cuando el cursor del ratón sale de los límites físicos de un objeto **Component**. El método **mouseDragged** se llama cuando el botón del ratón se oprime y se suelta, y el ratón se mueve (un proceso conocido como *arrastrar*). El evento **mouseDragged** va después de un evento **mousePressed** y antes de un evento **mouseReleased**. El método **mouseMoved** se llama cuando el ratón se mueve y el cursor del ratón está sobre un componente (y los botones del ratón no están oprimidos).



Observación de apariencia visual 29.10

Las llamadas al método **mouseDragged** se envían al objeto **MouseMotionListener** para el objeto **Component** en el que se inició la operación de arrastre. De manera similar, la llamada al método **mouseReleased** se envía al objeto **MouseListener** para el objeto **Component** en el que se inició la operación de arrastre.

La aplicación **RastreadorRaton** (figura 29.15) muestra el uso de los métodos **MouseListener** y **MouseMotionListener**. La clase de la aplicación implementa ambas interfaces, de manera que pueda escuchar sus propios eventos de ratón. Observe que el programador debe definir los siete métodos de estas dos interfaces cuando una clase implementa ambas interfaces.

```
1 // Figura 29.15: RastreadorRaton.java
2 // Demostración de los eventos de ratón.
3
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.swing.*;
7
8 public class RastreadorRaton extends JFrame
9         implements MouseListener, MouseMotionListener {
10    private JLabel barraEstado;
11
12    public RastreadorRaton()
13    {
14        super( "Demostracion de los eventos de raton" );
15
16        barraEstado = new JLabel();
17        getContentPane().add( barraEstado, BorderLayout.SOUTH );
18
19        // la aplicación escucha sus propios eventos de ratón
20        addMouseListener( this );
21        addMouseMotionListener( this );
22
23        setSize( 275, 100 );
24        show();
25    } // fin del constructor de RastreadorRaton
26
27    // Manejadores de eventos de MouseListener
28    public void mouseClicked( MouseEvent e )
29    {
30        barraEstado.setText( "Se hizo clic en [ " + e.getX() +
31                            ", " + e.getY() + " ]" );
32    } // fin del método mouseClicked
33
34    public void mousePressed( MouseEvent e )
35    {
36        barraEstado.setText( "Se oprimio en [ " + e.getX() +
37                            ", " + e.getY() + " ]" );
38    } // fin del método mousePressed
39
40    public void mouseReleased( MouseEvent e )
41    {
42        barraEstado.setText( "Se solto en [ " + e.getX() +
43                            ", " + e.getY() + " ]" );
44    } // fin del método mouseReleased
45
46    public void mouseEntered( MouseEvent e )
47    {
48        barraEstado.setText( "Raton dentro de la ventana" );
49    } // fin del método mouseEntered
50
51    public void mouseExited( MouseEvent e )
52    {
53        barraEstado.setText( "Raton fuera de la ventana" );
```

Figura 29.15 Demostración del manejo de eventos de ratón; **RastreadorRaton.java**.
(Parte 1 de 2.)

```

54     } // fin del método mouseExited
55
56     // Manejadores de eventos de MouseMotionListener
57     public void mouseDragged( MouseEvent e )
58     {
59         barraEstado.setText( "Se arrastro en [ " + e.getX() +
60                         ", " + e.getY() + " ]" );
61     } // fin del método mouseDragged
62
63     public void mouseMoved( MouseEvent e )
64     {
65         barraEstado.setText( "Se movio en [ " + e.getX() +
66                         ", " + e.getY() + " ]" );
67     } // fin del método mouseMoved
68
69     public static void main( String args[] )
70     {
71         RastreadorRaton ap = new RastreadorRaton();
72
73         ap.addWindowListener(
74             new WindowAdapter()
75             {
76                 public void windowClosing( WindowEvent e )
77                 {
78                     System.exit( 0 );
79                 } // fin del método windowClosing
80             } // fin de la clase interna anónima
81         ); // fin de addWindowListener
82     } // fin de main
82 } // fin de la clase RastreadorRaton

```

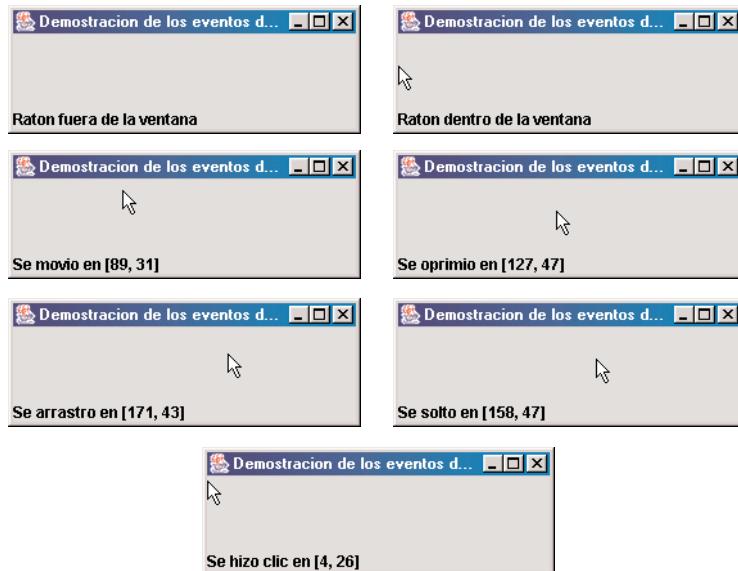


Figura 29.15 Demostración del manejo de eventos de ratón; **RastreadorRaton.java**. (Parte 2 de 2.)

Cada evento de ratón hace que aparezca un objeto **String** en el objeto **barraEstado** de **JLabel**, que se encuentra en la parte inferior de la ventana.

Las líneas 16 y 17 del constructor

```

barraEstado = new JLabel();
getContentPane().add( barraEstado, BorderLayout.SOUTH );

```

definen el objeto **barraEstado** de **JLabel** y lo adjuntan al panel de contenido. Hasta el momento, cada vez que utilizábamos el panel de contenido, se hacía una llamada al método **setLayout** para establecer en **FlowLayout** al administrador de diseño del panel de contenido. Esto permitía al panel de contenido mostrar de izquierda a derecha los componentes GUI que le íbamos adjuntando. Si los componentes GUI no caben en una sola línea, el diseño **FlowLayout** crea líneas adicionales para seguir mostrando los componentes GUI. En realidad, el administrador de diseño predeterminado es **BorderLayout**, el cual divide el área del panel de contenido en cinco regiones: norte, sur, este, oeste y centro. La línea 17 utiliza una nueva versión del método **add** de **Container** para adjuntar **barraEstado** a la región sur (**BorderLayout.SOUTH**), la cual se extiende a lo largo de toda la parte inferior del panel de contenido. Más adelante en este capítulo describiremos detalladamente el uso de **BorderLayout** y varios otros administradores de diseño.

Las líneas 20 y 21 del constructor

```
addMouseListener( this );
addMouseMotionListener( this );
```

registran el objeto de ventana **RastreadorRaton** como el componente que escucha sus propios eventos de ratón. Los métodos **addMouseListener** y **addMouseMotionListener** son métodos de **Component** que pueden utilizarse para registrar componentes para escuchar eventos de ratón de un objeto de cualquier clase que extienda a **Component**.

Cuando el ratón entra o sale del área de la aplicación, se llama a los métodos **mouseEntered** (línea 46) y **mouseExited** (línea 51), respectivamente. Ambos métodos muestran un mensaje en la **barraEstado**, el cual indica que el ratón se encuentra dentro de la aplicación, o que está fuera de ella (vea las primeras dos capturas de imágenes de pantalla).

Cuando ocurre cualquiera de los otros cinco eventos, aparece un mensaje en la **barraEstado** que incluye un objeto **String**, el cual representa el evento que ocurrió y las coordenadas en donde ocurrió ese evento de ratón. Las coordenadas **x** y **y** del ratón, al momento en que ocurrió el evento, se obtienen mediante los métodos **getX** y **getY** de **MouseEvent**, respectivamente.

29.11 Administradores de diseño

Los *administradores de diseño* ordenan los componentes GUI en un contenedor, para fines de presentación. Los administradores de diseño proporcionan herramientas de diseño básicas, que son más fáciles de utilizar que determinar la posición y el tamaño exactos de cada componente GUI. Esto permite al programador concentrarse en la “apariencia visual” básica, para dejar a los administradores de diseño que procesen la mayor parte de los detalles de diseño.

Observación de apariencia visual 29.11



La mayoría de los entornos de programación de Java proporcionan herramientas de diseño GUI, las cuales ayudan a un programador a diseñar de manera gráfica una GUI, y después escriben automáticamente el código de Java necesario para crear la GUI.

Algunos diseñadores de GUIs también permiten al programador utilizar los administradores de diseño que describimos aquí. La figura 29.16 sintetiza los administradores de diseño que presentamos en este capítulo.

Administrador de diseño Descripción

FlowLayout	Es el predeterminado para java.awt.Applet , java.awt.Panel y javax.swing.JPanel . Coloca los componentes secuencialmente (de izquierda a derecha) en el orden en el que se agregaron. También es posible especificar el orden de los componentes utilizando el método add de Container , el cual toma como argumentos un objeto Component y un valor entero que representa la posición del índice.
BorderLayout	Es el predeterminado para los paneles de contenido de objetos JFrame (y otras ventanas) y JApplet . Ordena los componentes en cinco áreas: Norte, Sur, Este, Oeste y Centro.
GridLayout	Ordena los componentes en filas y columnas.

Figura 29.16 Administradores de diseño.

La mayoría de los ejemplos anteriores de applets y aplicaciones en los que creamos nuestra propia GUI utilizan el administrador de diseño **FlowLayout**. La clase **FlowLayout** hereda de la clase **Object** e implementa la interfaz **LayoutManager**, la cual define los métodos que utiliza un administrador de diseño para ordenar y ajustar el tamaño de los componentes GUI en un contenedor.

29.11.1 FlowLayout

Éste es el administrador de diseño más básico. Los componentes GUI se colocan en un contenedor de izquierda a derecha, en el orden en el que se agregan al contenedor. Al llegar al límite del contenedor, los componentes continúan en la siguiente línea. La clase **FlowLayout** permite que los componentes GUI estén *alineados a la izquierda, centrados* (la opción predeterminada) y *alineados a la derecha*.

La aplicación de la figura 29.17 crea tres objetos **JButton** y los agrega a la aplicación utilizando el administrador de diseño **FlowLayout**. Los componentes se alinean automáticamente al centro. Cuando el usuario hace clic en **Izquierda**, la alineación del administrador de diseño cambia a un **FlowLayout** con alineación a la izquierda. Cuando el usuario hace clic en **Derecha**, la alineación del administrador de diseño cambia a un **FlowLayout** con alineación a la derecha. Cuando el usuario hace clic en **Centro**, la alineación del administrador de diseño cambia a un **FlowLayout** con alineación al centro. Cada botón tiene su propio manejador de eventos que se define mediante una clase interna que implementa a **ActionListener**.

```
1 // Figura 29.17: DemoFlowLayout.java
2 // Demostración de las alineaciones de FlowLayout.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class DemoFlowLayout extends JFrame {
8     private JButton izquierda, centro, derecha;
9     private Container c;
10    private FlowLayout disenio;
11
12    public DemoFlowLayout()
13    {
14        super( "Demo de FlowLayout" );
15
16        disenio = new FlowLayout();
17
18        c = getContentPane();
19        c.setLayout( disenio );
20
21        izquierda = new JButton( "Izquierda" );
22        izquierda.addActionListener(
23            new ActionListener() {
24                public void actionPerformed( ActionEvent e )
25                {
26                    disenio.setAlignment( FlowLayout.LEFT );
27
28                    // vuelve a alinear los componentes adjuntos
29                    disenio.layoutContainer( c );
30                } // fin del método actionPerformed
31            } // fin de la clase interna anónima
32        ); // fin de addActionListener
33        c.add( izquierda );
```

Figura 29.17 Programa que demuestra el uso de componentes en **FlowLayout**; **DemoFlowLayout.java**. (Parte 1 de 2.)

```

34
35     centro = new JButton( "Centro" );
36     centro.addActionListener(
37         new ActionListener() {
38             public void actionPerformed( ActionEvent e )
39             {
40                 disenio.setAlignment( FlowLayout.CENTER );
41
42                 // volver a alinear los componentes adjuntos
43                 disenio.setLayoutContainer( c );
44             } // fin del método actionPerformed
45         } // fin de la clase interna anónima
46     ); // fin de addActionListener
47     c.add( centro );
48
49     derecha = new JButton( "Derecha" );
50     derecha.addActionListener(
51         new ActionListener() {
52             public void actionPerformed( ActionEvent e )
53             {
54                 disenio.setAlignment( FlowLayout.RIGHT );
55
56                 // vuelve a alinear los componentes adjuntos
57                 disenio.setLayoutContainer( c );
58             } // fin del método actionPerformed
59         } // fin de la clase interna anónima
60     ); // fin de addActionListener
61     c.add( derecha );
62
63     setSize( 300, 75 );
64     show();
65 } // fin del constructor de DemoFlowLayout
66
67 public static void main( String args[] )
68 {
69     DemoFlowLayout ap = new DemoFlowLayout();
70
71     ap.addWindowListener(
72         new WindowAdapter() {
73             public void windowClosing( WindowEvent e )
74             {
75                 System.exit( 0 );
76             } // fin del método windowClosing
77         } // fin de la clase interna anónima
78     ); // fin de addWindowListener
79 }
80 } // fin de la clase DemoFlowLayout

```



Figura 29.17 Programa que demuestra el uso de componentes en **FlowLayout**; **DemoFlowLayout.java**. (Parte 2 de 2.)

Como vimos anteriormente, el diseño de un contenedor se establece mediante el método ***setLayout*** de la clase **Container**. La línea 19

```
c.setLayout( disenio );
```

establece el administrador de diseño del panel de contenido al **FlowLayout** definido en la línea 16. En general, el diseño se establece antes de agregar cualquier componente GUI a un contenedor.

Observación de apariencia visual 29.12



Cada contenedor puede tener solamente un administrador de diseño a la vez (varios contenedores en el mismo programa pueden tener distintos administradores de diseño).

El manejador de eventos ***actionPerformed*** de cada botón ejecuta dos instrucciones. Por ejemplo, la línea 26 del método ***actionPerformed*** del botón **izquierda**

```
disenio.setAlignment( FlowLayout.LEFT );
```

utiliza el método ***setAlignment*** de **FlowLayout** para cambiar la alineación de **FlowLayout** a la izquierda (**FlowLayout.LEFT**). La línea 29

```
disenio.setLayoutContainer( c );
```

utiliza el método ***LayoutContainer*** de la interfaz **LayoutManager** para especificar que el panel de contenido debe volver a ordenarse con base en el diseño ajustado.

De acuerdo con el botón en el que se haya hecho clic, el método ***actionPerformed*** de cada botón establece la alineación de **FlowLayout** a **FlowLayout.LEFT**, **FlowLayout.CENTER** o **FlowLayout.RIGHT**.

29.11.2 BorderLayout

El administrador de diseño **BorderLayout** (el predeterminado para el panel de contenido) ordena los componentes en cinco regiones: *Norte*, *Sur*, *Este*, *Oeste* y *Centro* (la región Norte corresponde a la parte superior del contenedor). La clase **BorderLayout** hereda de **Object** e implementa la interfaz **LayoutManager2** (una subinterfaz de **LayoutManager** que agrega varios métodos para mejorar el procesamiento de los diseños).

Es posible agregar hasta cinco componentes directamente a un diseño **BorderLayout**; uno para cada región. Los componentes que se colocan en las regiones Norte y Sur se extienden horizontalmente hacia los lados del contenedor, y su altura depende de los componentes que se coloquen en esas regiones. Las regiones Este y Oeste se expanden verticalmente entre las regiones Norte y Sur, y su ancho depende de los componentes que se coloquen en esas regiones. El componente colocado en la región Centro se expande para ocupar todo el espacio restante en el diseño (ésta es la razón por la que el objeto **JTextArea** de la figura 29.18 ocupa la ventana completa). Si las cinco regiones están ocupadas, todo el espacio del contenedor se cubre con componentes GUI. Si la región Norte o la región Sur no están ocupadas, los componentes GUI de las regiones Este, Centro y Oeste se expanden verticalmente para llenar el espacio restante. Si la región Este o la región Oeste no están ocupadas, el componente GUI de la región Centro se expande horizontalmente para llenar el espacio restante. Si la región Centro no está ocupada, el área se deja vacía; los demás componentes GUI no se expanden para llenar el espacio restante.

La aplicación de la figura 29.18 demuestra el uso del administrador de diseño **BorderLayout** con cinco objetos **JButton**.

```

1 // Figura 29.18: DemoBorderLayout.java
2 // Demostración de BorderLayout.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
```

Figura 29.18 Demostración del uso de componentes en **BorderLayout**; **DemoBorderLayout.java**. (Parte 1 de 3.)

```
7  public class DemoBorderLayout extends JFrame
8                      implements ActionListener {
9      private JButton b[];
10     private String nombres[] =
11         { "Ocultar Norte", "Ocultar Sur", "Ocultar Este",
12           "Ocultar Oeste", "Ocultar Centro" };
13     private BorderLayout disenio;
14
15    public DemoBorderLayout()
16    {
17        super( "Demo de BorderLayout" );
18
19        disenio = new BorderLayout( 5, 5 );
20
21        Container c = getContentPane();
22        c.setLayout( disenio );
23
24        // instanciar objetos botón
25        b = new JButton[ nombres.length ];
26
27        for ( int i = 0; i < nombres.length; i++ ) {
28            b[ i ] = new JButton( nombres[ i ] );
29            b[ i ].addActionListener( this );
30        } // fin de for
31
32        // el orden no importa
33        c.add( b[ 0 ], BorderLayout.NORTH ); // Posición Norte
34        c.add( b[ 1 ], BorderLayout.SOUTH ); // Posición Sur
35        c.add( b[ 2 ], BorderLayout.EAST ); // Posición Este
36        c.add( b[ 3 ], BorderLayout.WEST ); // Posición Oeste
37        c.add( b[ 4 ], BorderLayout.CENTER ); // Posición Centro
38
39        setSize( 400, 250 );
40        show();
41    } // fin del constructor DemoBorderLayout
42
43    public void actionPerformed( ActionEvent e )
44    {
45        for ( int i = 0; i < b.length; i++ )
46            if ( e.getSource() == b[ i ] )
47                b[ i ].setVisible( false );
48            else
49                b[ i ].setVisible( true );
50
51        // reajusta el diseño del panel de contenido
52        disenio.layoutContainer( getContentPane() );
53    } // fin del método actionPerformed
54
55    public static void main( String args[] )
56    {
57        DemoBorderLayout ap = new DemoBorderLayout();
58
59        ap.addWindowListener(
60            new WindowAdapter() {
```

Figura 29.18 Demostración del uso de componentes en **BorderLayout**; **DemoBorderLayout.java**. (Parte 2 de 3.)

```

61         public void windowClosing( WindowEvent e )
62     {
63         System.exit( 0 );
64     } // fin del método windowClosing
65 } // fin de la clase interna anónima
66 ); // fin de addWindowListener
67 } // fin de main
68 } // fin de la clase DemoBorderLayout

```

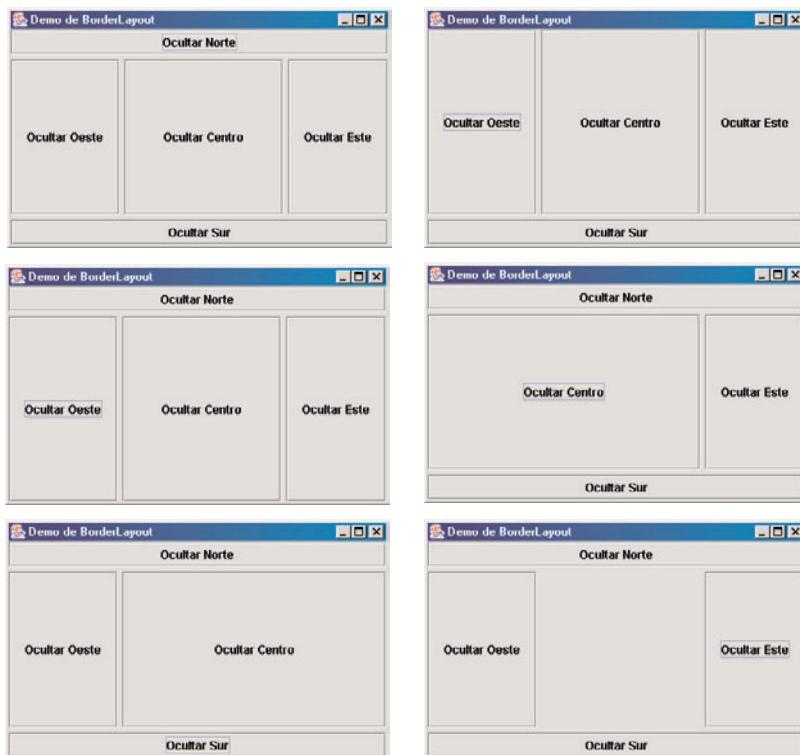


Figura 29.18 Demostración del uso de componentes en **BorderLayout**; **DemoBorderLayout.java**. (Parte 3 de 3.)

La línea 19 del constructor

```
disenio = new BorderLayout( 5, 5 );
```

define un diseño **BorderLayout**. Los argumentos especifican el número de píxeles entre componentes ordenados horizontalmente (*espacio libre horizontal*) y el número de píxeles entre componentes ordenados verticalmente (*espacio libre vertical*), respectivamente. El constructor predeterminado de **BorderLayout** proporciona 0 píxeles de espacio libre horizontal y vertical. La línea 22 utiliza el método **setLayout** para establecer el diseño del panel de contenido a **disenio**.

Para agregar objetos **Component** a un diseño **BorderLayout** se requiere un método **add** distinto de la clase **Container**, el cual toma dos argumentos: el objeto **Component** que va a agregarse y la región en la que se colocará este objeto. Por ejemplo, la línea 33

```
add( b[ 0 ], BorderLayout.NORTH ); // Posición Norte
```

especifica que el componente **b[0]** va a colocarse en la posición **NORTH**. Los componentes pueden agregarse en cualquier orden, pero solamente puede agregarse un componente a cada región.

Observación de apariencia visual 29.13



*Si no se especifica una región al agregar un objeto **Component** a un diseño **BorderLayout**, se asume que el objeto **Component** va a agregarse a la región **BorderLayout.CENTER**.*

Error común de programación 29.5



*Si se agrega más de un componente a una región específica en un diseño **BorderLayout**, sólo se desplegará el último componente que se haya agregado. No hay un mensaje de error para indicar este problema.*

Cuando el usuario hace clic en un objeto **JButton** específico del diseño, se hace una llamada al método **actionPerformed** (línea 43). El ciclo **for** de la línea 46 utiliza la siguiente estructura **if/else**

```
if ( e.getSource() == b[ i ] )
    b[ i ].setVisible( false );
else
    b[ i ].setVisible( true );
```

para ocultar el objeto **JButton** específico que generó el evento. El método **setVisible** (heredado por **JButton** de la clase **Component**) se llama con un argumento **false** para ocultar el objeto **JButton**. Si el objeto **JButton** actual del arreglo no es el que generó el evento, se hace una llamada al método **setVisible** con un argumento **true** para garantizar que el objeto **JButton** se despliegue en la pantalla. La línea 52

```
disenio.setLayoutContainer( getContentPane() );
```

utiliza el método **layoutContainer** de **LayoutManager** para recalcular el diseño del panel de contenido. Observe en las capturas de pantalla de la figura 29.18 que ciertas regiones del diseño **BorderLayout** cambian de forma, a medida que se ocultan y se despliegan los objetos **JButton** en otras regiones. Intente cambiar el tamaño de la ventana de la aplicación para que vea cómo se ajusta el tamaño de las diversas regiones, con base en el ancho y la altura de la ventana.

29.11.3 GridLayout

El administrador de diseño **GridLayout** divide el contenedor en una cuadrícula, de manera que los componentes puedan colocarse en filas y columnas. La clase **GridLayout** hereda directamente de la clase **Object** e implementa la interfaz **LayoutManager**. Cada objeto **Component** de un diseño **GridLayout** tiene el mismo ancho y alto. Los componentes se agregan a un diseño **GridLayout** a partir de la celda superior izquierda de la cuadrícula, y continúan agregándose de izquierda a derecha hasta que la fila se llena. Después, el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente. La figura 29.19 muestra el uso del administrador de diseño **GridLayout** con seis objetos **JButton**.

```
1 // Figura 29.19: DemoGridLayout.java
2 // Demostración de GridLayout.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class DemoGridLayout extends JFrame
8         implements ActionListener {
9     private JButton b[];
10    private String nombres[] =
11        { "uno", "dos", "tres", "cuatro", "cinco", "seis" };
12    private boolean alternar = true;
13    private Container c;
14    private GridLayout cuadricula1, cuadricula2;
15
16    public DemoGridLayout()
```

Figura 29.19 Programa que muestra el uso de componentes en **GridLayout**; **DemoGridLayout.java**. (Parte 1 de 2.)

```

17     {
18         super( "Demostracion de GridLayout" );
19
20         cuadricula1 = new GridLayout( 2, 3, 5, 5 );
21         cuadricula2 = new GridLayout( 3, 2 );
22
23         c = getContentPane();
24         c.setLayout( cuadricula1 );
25
26         // crea y agrega botones
27         b = new JButton[ nombres.length ];
28
29         for ( int i = 0; i < nombres.length; i++ ) {
30             b[ i ] = new JButton( nombres[ i ] );
31             b[ i ].addActionListener( this );
32             c.add( b[ i ] );
33         }
34
35         setSize( 300, 150 );
36         show();
37     } // fin del constructor de DemoGridLayout
38
39     public void actionPerformed( ActionEvent e )
40     {
41         if ( alternar )
42             c.setLayout( cuadricula2 );
43         else
44             c.setLayout( cuadricula1 );
45
46         alternar = !alternar;
47         c.validate();
48     } // fin del método actionPerformed
49
50     public static void main( String args[] )
51     {
52         DemoGridLayout ap = new DemoGridLayout();
53
54         ap.addWindowListener(
55             new WindowAdapter() {
56                 public void windowClosing( WindowEvent e )
57                 {
58                     System.exit( 0 );
59                 } // fin del método windowClosing
60             } // fin de la clase interna anónima
61         ); // fin de addWindowListener
62     } // fin de main
63 } // fin de la clase DemoGridLayout

```

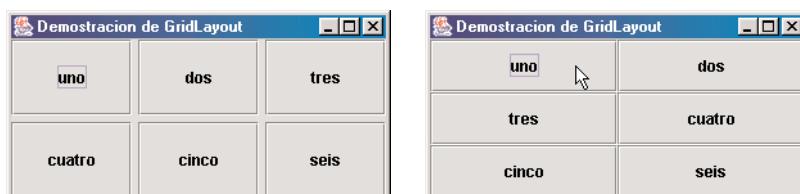


Figura 29.19 Programa que muestra el uso de componentes en **GridLayout**; **DemoGridLayout.java**. (Parte 2 de 2.)

Las líneas 20 y 21 del constructor

```
cuadricula1 = new GridLayout( 2, 3, 5, 5 );
cuadricula2 = new GridLayout( 3, 2 );
```

definen dos objetos **GridLayout**. El constructor de **GridLayout** utilizado en la línea 20 especifica un objeto **GridLayout** con 2 filas, 3 columnas, 5 pixeles de espacio libre horizontal entre los objetos **Component** de la cuadrícula, y 5 pixeles de espacio libre vertical entre los objetos **Component** de la cuadrícula. El constructor de **GridLayout** utilizado en la línea 21 especifica un objeto **GridLayout** con 3 filas, 2 columnas y nada de espacio libre.

Los objetos **JButton** de este ejemplo se ordenan inicialmente por medio de **cuadricula1** (que se establece para el panel de contenido en la línea 24 a través del método **setLayout**). El primer componente se agrega a la primera columna de la primera fila. El siguiente componente se agrega a la segunda columna de la primera fila, etcétera. Cuando se oprime un objeto **JButton**, se hace una llamada al método **actionPerformed** (línea 39). Cada llamada a **actionPerformed** cambia el diseño entre **cuadricula2** y **cuadricula1**.

La línea 47

```
c.validate();
```

muestra una manera de redistribuir un contenedor que haya cambiado su diseño. El método **validate** de **Container** recalcula la distribución del contenedor con base en el administrador de diseño actual para el objeto **Container** y el conjunto actual de componentes GUI desplegados en pantalla.

29.12 Paneles

Las GUIs complejas (como la figura 29.1) requieren que cada componente se coloque en una ubicación exacta. A menudo, éstas consisten en varios *paneles* en los que cada componente está ordenado con un diseño específico. Los paneles se crean mediante la clase **JPanel** (una subclase de **JComponent**). La clase **JComponent** hereda de **java.awt.Container**, por lo que todo **JPanel** es un **Container**. Por lo tanto, es posible agregar muchos componentes a los objetos **JPanel**, incluso otros paneles.

El programa de la figura 29.20 muestra cómo puede utilizarse un objeto **JPanel** para crear un diseño más complejo para objetos **Component**.

```

1 // Figura 29.20: DemoPanel.java
2 // Uso de un objeto JPanel para ayudar a distribuir los componentes en un
3 // diseño.
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.swing.*;
7
8 public class DemoPanel extends JFrame {
9     private JPanel panelBotones;
10    private JButton botones[];
11
12    public DemoPanel()
13    {
14        super( "Demostracion de JPanel" );
15
16        Container c = getContentPane();
17        panelBotones = new JPanel();
18        botones = new JButton[ 5 ];
19        panelBotones.setLayout(

```

Figura 29.20 Un objeto **JPanel** con cinco objetos **JButton** en un diseño **GridLayout** adjunto a la región **SOUTH** de un diseño **BorderLayout**; **DemoPanel.java**. (Parte 1 de 2.)

```

20         new GridLayout( 1, botones.length ) );
21
22     for ( int i = 0; i < botones.length; i++ ) {
23         botones[ i ] = new JButton( "Boton " + (i + 1) );
24         panelBotones.add( botones[ i ] );
25     }
26
27     c.add( panelBotones, BorderLayout.SOUTH );
28
29     setSize( 425, 150 );
30     show();
31 } // fin del constructor DemoPanel
32
33 public static void main( String args[] )
34 {
35     DemoPanel ap = new DemoPanel();
36
37     ap.addWindowListener(
38         new WindowAdapter() {
39             public void windowClosing( WindowEvent e )
40             {
41                 System.exit( 0 );
42             } // fin del método windowClosing
43         } // fin de la clase interna anónima
44     ); // fin de addWindowListener
45 } // fin de main
46 } // fin de la clase DemoPanel

```



Figura 29.20 Un objeto **JPanel** con cinco objetos **JButton** en un diseño **GridLayout** adjunto a la región **SOUTH** de un diseño **BorderLayout**; **DemoPanel.java**. (Parte 2 de 2.)

Una vez creado el objeto **panelBotones** de **JPanel** de la línea 16, las líneas 19 y 20

```

panelBotones.setLayout(
    new GridLayout( 1, botones.length ) );

```

establecen el diseño de **panelBotones** en un **GridLayout** de una fila y cinco columnas (hay cinco objetos **JButton** en el arreglo **botones**). Los cinco objetos **JButton** del arreglo **botones** se agregan al objeto **JPanel** en el ciclo de la línea 24, por medio de la instrucción:

```

panelBotones.add( botones[ i ] );

```

Observe que los botones se agregan directamente al objeto **JPanel**; la clase **JPanel** no tiene un panel de contenido como el de un applet o un objeto **JFrame**. La línea 27

```

c.add( panelBotones, BorderLayout.SOUTH );

```

utiliza el diseño **BorderLayout** predeterminado del panel de contenido para agregar **panelBotones** a la región **SOUTH**. Observe que la altura de esta región se rige por los botones de **panelBotones**. Un objeto **JPanel** ajusta su tamaño según los componentes que contiene. A medida que se agregan más componentes, el objeto **JPanel** crece (de acuerdo con las restricciones de su administrador de diseño) para dar cabida a esos componentes. Ajuste el tamaño de la ventana para que vea cómo el administrador de diseño afecta al tamaño de los objetos **JButton**.

29.13 Creación de una subclase autocontenido de JPanel

Un objeto **JPanel** puede utilizarse como *área de dibujo dedicada*, la cual puede recibir eventos de ratón y, a menudo, se extiende para crear nuevos componentes. En ejercicios anteriores tal vez haya observado que el combinar componentes GUI de Swing con el dibujo en una ventana o subprograma, con frecuencia ocasiona que los componentes GUI o los gráficos se desplieguen en forma incorrecta. Esto se debe a que los componentes GUI de Swing se despliegan utilizando las mismas técnicas de gráficos que los dibujos, y se despliegan en la misma área que los dibujos. El orden en el que se despliegan los componentes GUI y en el que se realiza el dibujo puede ocasionar que se dibuje sobre los componentes GUI, o que los componentes GUI cubran parte de los gráficos. Para solucionar este problema, podemos separar la GUI y los gráficos, creando áreas de dibujo dedicadas como subclases de **JPanel**.

Observación de apariencia visual 29.14



Combinar gráficos y componentes GUI puede ocasionar un despliegue incorrecto de los gráficos, de los componentes GUI o de ambos. Utilizar objetos JPanel para dibujar puede eliminar este problema, proporcionando un área de dibujo dedicada para los gráficos.

Los componentes Swing que heredan de la clase **JComponent** contienen el método **paintComponent**, el cual les ayuda a dibujar correctamente dentro del contexto de una GUI de Swing. Al personalizar un objeto **JPanel** para usarlo como área de dibujo dedicada, el método **paintComponent** debe redefinirse de la siguiente manera:

```
public void paintComponent( Graphics g )
{
    super.paintComponent( g );
    // código adicional para dibujar
}
```

Observe que la llamada a la versión de **paintComponent** correspondiente a la superclase aparece como la primera instrucción en el cuerpo del método redefinido. Esto garantiza que la acción de dibujar ocurra en el orden adecuado y que el mecanismo de dibujo de Swing permanezca intacto. Si no se hace la llamada a la versión de **paintComponent** correspondiente a la superclase, por lo general lo que ocurre es que el componente GUI personalizado (en este caso, la subclase de **JPanel**) no se desplegará apropiadamente en la interfaz de usuario. Además, si se hace la llamada a la versión de la superclase después de ejecutar las instrucciones de dibujo personalizadas, los resultados normalmente se borran.

Observación de apariencia visual 29.15



Cuando se redefine el método paintComponent de un objeto JComponent, la primera instrucción del cuerpo siempre debe ser una llamada a la versión original del método de la superclase.

Error común de programación 29.6



Cuando se redefine el método paintComponent de un objeto JComponent, si no se hace una llamada a la versión original de paintComponent de la superclase, el componente GUI no podrá desplegarse apropiadamente en la GUI.

Error común de programación 29.7



Cuando se redefine el método paintComponent de un objeto JComponent, al llamar a la versión original de paintComponent de la superclase después de realizar otro dibujo, se borran los demás dibujos.

Las clases **JFrame** y **JApplet** no son subclases de **JComponent**; por lo tanto, no contienen el método **paintComponent**. Para dibujar directamente en subclases de **JFrame** y **JApplet**, debe redefinir el método **paint**.

Observación de apariencia visual 29.16



Llamar a repaint para un componente GUI de Swing indica que ese componente debe pintarse lo más pronto posible. El fondo del componente GUI se borra solamente si el componente es opaco. La mayoría de los componentes Swing son transparentes de manera predeterminada. Es posible pasar un argumento booleano al método setOpaque de JComponent para indicar si el componente es opaco (true), o transparente (false). Los componentes GUI del paquete java.awt son distintos de los componentes Swing en cuanto a que repaint produce una llamada al método update de Component (con lo cual se borra el fondo del componente), y update, a su vez, llama al método paint (en lugar de llamar a paintComponent).

Los objetos **JPanel** no generan eventos convencionales como los botones, campos de texto y ventanas, pero son capaces de reconocer eventos de menor nivel, tales como los eventos de ratón y de tecla. El programa de la figura 29.21 permite al usuario dibujar un óvalo en una subclase de **JPanel** con el ratón. La clase **PanelAutoContenido** escucha sus propios eventos de ratón y dibuja un óvalo sobre sí misma. Para determinar la ubicación y el tamaño del óvalo, el usuario debe oprimir el botón del ratón y mantenerlo así, arrastrarlo y soltarlo. La clase **PanelAutoContenido** se encuentra en el paquete **com.deitel.cpec4.cap29**, para poder reutilizarla en el futuro. Por esta razón se importa (mediante la instrucción **import** de la línea 7) a la clase de la aplicación **PanelAutoContenido**.

```
1 // Figura 29.21: PruebaPanelAutoContenido.java
2 // Creación de una subclase autocontenido de JPanel
3 // que procesa sus propios eventos de ratón.
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.swing.*;
7 import com.deitel.cpec4.cap29.PanelAutoContenido;
8
9 public class PruebaPanelAutoContenido extends JFrame {
10     private PanelAutoContenido miPanel;
11
12     public PruebaPanelAutoContenido()
13     {
14         miPanel = new PanelAutoContenido();
15         miPanel.setBackground( Color.yellow );
16
17         Container c = getContentPane();
18         c.setLayout( new FlowLayout() );
19         c.add( miPanel );
20
21         addMouseMotionListener(
22             new MouseMotionListener() {
23                 public void mouseDragged( MouseEvent e )
24                 {
25                     setTitle( "Arrastrando: x=" + e.getX() +
26                             "; y=" + e.getY() );
27                 } // fin del método mouseDragged
28
29                 public void mouseMoved( MouseEvent e )
30                 {
31                     setTitle( "Moviendo: x=" + e.getX() +
32                             "; y=" + e.getY() );
33                 } // fin del método mouseMoved
34             } // fin de la clase interna anónima
35         ); // fin de addMouseMotionListener
36
37         setSize( 300, 200 );
38         show();
39     } // fin del constructor PruebaPanelAutoContenido
40
41     public static void main( String args[] )
42     {
43         PruebaPanelAutoContenido ap =
44             new PruebaPanelAutoContenido();
```

Figura 29.21 Cómo capturar eventos de ratón con un objeto **JPanel**; **PruebaPanelAutoContenido.java**. (Parte 1 de 2)

```

45
46     ap.addWindowListener(
47         new WindowAdapter() {
48             public void windowClosing( WindowEvent e )
49             {
50                 System.exit( 0 );
51             } // fin del método windowClosing
52         } // fin de la clase interna anónima
53     ); // fin de addWindowListener
54 } // fin de main
55 } // fin de la clase PruebaPanelAutoContenido

```

Figura 29.21 Cómo capturar eventos de ratón con un objeto **JPanel**; **PruebaPanelAutoContenido.java**. (Parte 2 de 2.)

```

56 // Figura 29.21: PanelAutoContenido.java
57 // Una clase JPanel autocontenida que
58 // maneja sus propios eventos de ratón.
59 package com.deitel.cpec4.cap29;
60
61 import java.awt.*;
62 import java.awt.event.*;
63 import javax.swing.*;
64
65 public class PanelAutoContenido extends JPanel {
66     private int x1, y1, x2, y2;
67
68     public PanelAutoContenido()
69     {
70         addMouseListener(
71             new MouseAdapter() {
72                 public void mousePressed( MouseEvent e )
73                 {
74                     x1 = e.getX();
75                     y1 = e.getY();
76                 } // fin del método mousePressed
77
78                 public void mouseReleased( MouseEvent e )
79                 {
80                     x2 = e.getX();
81                     y2 = e.getY();
82                     repaint();
83                 } // fin del método mouseReleased
84             } // fin de la clase interna anónima
85         ); // fin de addMouseListener
86
87         addMouseMotionListener(
88             new MouseMotionAdapter() {
89                 public void mouseDragged( MouseEvent e )
90                 {
91                     x2 = e.getX();
92                     y2 = e.getY();
93                     repaint();

```

Figura 29.21 Cómo capturar eventos de ratón con un objeto **JPanel**; **PanelAutoContenido.java**. (Parte 1 de 2.)

```

94             } // fin del método mouseDragged
95         } // fin de la clase interna anónima
96     ); // fin de addMouseMotionListener
97 } // fin del constructor PanelAutoContenido
98
99     public Dimension getPreferredSize()
100    {
101        return new Dimension( 150, 100 );
102    } // fin del método getPreferredSize
103
104    public void paintComponent( Graphics g )
105    {
106        super.paintComponent( g );
107
108        g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
109                    Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
110    } // fin del método paintComponent
111 } // fin de la clase PanelAutoContenido

```

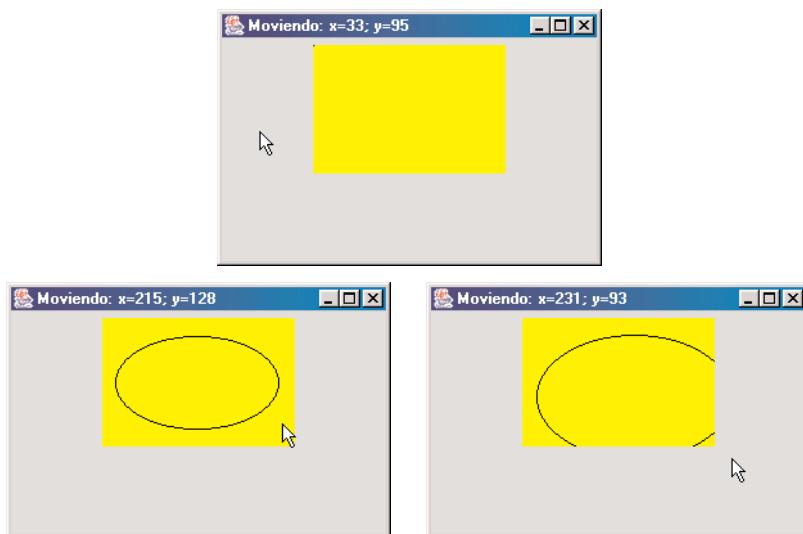


Figura 29.21 Cómo capturar eventos de ratón con un objeto **JPanel**; **PanelAutoContenido.java**. (Parte 2 de 2.)

El método constructor (línea 12) de la clase de la aplicación **PruebaPanelAutoContenido** crea una instancia de la clase **PanelAutoContenido** y establece en amarillo el color de fondo del **PanelAutoContenido**, de manera que su área sea visible y contraste con el fondo de la ventana de la aplicación.

Para que podamos demostrar la diferencia entre los eventos de movimiento del ratón en el **PanelAutoContenido** y los eventos de movimiento del ratón en la ventana de la aplicación, las líneas 21 a 35 crean una clase interna anónima para manejar los eventos de movimiento del ratón en la aplicación. Los manejadores de eventos **mouseDragged** y **mouseMoved** utilizan el método **setTitle** (heredado de la clase **java.awt.Frame**) para mostrar un objeto **String** en la barra de título de la ventana, e indican las coordenadas *x* y *y* en donde ocurrió el evento de movimiento del ratón.

La clase **PanelAutoContenido** (líneas 65 a 111) extiende a la clase **JPanel**. Las variables de instancia **x1** y **y1** almacenan las coordenadas iniciales en donde ocurre el evento **mousePressed** en el **PanelAutoContenido**. Las variables de instancia **x2** y **y2** almacenan las coordenadas en donde el usuario arrastra el ratón o suelta el botón de éste. Todas las coordenadas son con respecto a la esquina superior izquierda del **PanelAutoContenido**.

Observación de apariencia visual 29.17



El proceso de dibujar en cualquier componente GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0, 0) de ese componente GUI.

El constructor **PanelAutoContenido** (línea 68) utiliza los métodos **addMouseListener** y **addMouseMotionListener** para registrar objetos de la clase interna anónima, para manejar los eventos del ratón y los eventos de movimiento del ratón para el **PanelAutoContenido**. En realidad, sólo se redefinen los métodos **mousePressed** (línea 72), **mouseReleased** (línea 78) y **mouseDragged** (línea 89) para realizar tareas. Los otros métodos manejadores de eventos de ratón se heredan de las clases **MouseAdapter** y **MouseMotionAdapter**, cuando se definen las clases internas anónimas.

Al extender la clase **JPanel**, en realidad creamos un nuevo componente GUI. A menudo, los administradores de diseño utilizan el método **getPreferredSize** de un componente GUI (heredado de la clase **java.awt.Component**) para determinar los mejores valores para el ancho y la altura de dicho componente cuando éste se diseña como parte de una GUI. Si un nuevo componente tiene un mejor ancho y altura, éste debe redefinir al método **getPreferredSize** (líneas 99 a 102) para devolver ese ancho y esa altura como objetos de la clase **Dimension** (del paquete **java.awt**).

Observación de apariencia visual 29.18



El tamaño predeterminado de un objeto JPanel es de 0 pixeles de ancho y de 0 pixeles de alto.

Observación de apariencia visual 29.19



Al crear subclases de JPanel (o de cualquier otro JComponent), se debe redefinir el método getPreferredSize si el nuevo componente debe tener mejores valores para el ancho y la altura.

El método **paintComponent** (línea 104) se redefine en la clase **PanelAutoContenido** para dibujar un óvalo. Para determinar el ancho, la altura y la esquina superior izquierda, el usuario debe oprimir el botón del ratón y mantenerlo así, y arrastrarlo y soltarlo en el área de dibujo del **PanelAutoContenido**.

Las coordenadas iniciales **x1** y **y1** en el área de dibujo del **PanelAutoContenido** se capturan en el método **mousePressed** (línea 72). A medida que el usuario arrastra el ratón después de la operación inicial en **mousePressed**, el programa genera una serie de llamadas a **mouseDragged** (línea 89) mientras el usuario continúa oprimiendo el botón del ratón y moviéndolo. Cada llamada captura en las variables **x2** y **y2** la posición actual del ratón con respecto a la esquina superior izquierda del **PanelAutoContenido**, y se hace una llamada a **repaint** para dibujar la versión actual del óvalo. La acción de dibujar queda confinada estrictamente al **PanelAutoContenido**, incluso si el usuario arrastra el ratón fuera del área de dibujo del **PanelAutoContenido**. Cualquier cosa que se dibuje fuera del **PanelAutoContenido** se *recorta*; los pixeles no se despliegan fuera de los límites del **PanelAutoContenido**.

Los cálculos proporcionados en el método **paintComponent** determinan la esquina superior izquierda apropiada utilizando dos veces el método **Math.min**, para encontrar el valor más pequeño para las coordenadas **x** y **y**. El ancho y la altura del óvalo deben ser valores positivos, pues de lo contrario éste no aparecerá en pantalla. El método **Math.abs** obtiene el valor absoluto de las restas **x1 - x2** y **y1 - y2** que determinan el ancho y la altura del rectángulo delimitador del óvalo, respectivamente. Cuando se completan los cálculos, **paintComponent** dibuja el óvalo. La llamada a la versión de **paintComponent** correspondiente a la superclase al principio del método garantiza que se borre el óvalo anterior mostrado en el **PanelAutoContenido**, antes de que el nuevo se despliegue en la pantalla.

Observación de apariencia visual 29.20



La mayoría de los componentes Swing pueden ser transparentes u opacos. Si un componente GUI de Swing es opaco, al llamar a su método paintComponent su fondo se borrará; en caso contrario, no se borrará.

Observación de apariencia visual 29.21



*La clase JComponent proporciona el método **setOpaque** que toma un argumento booleano para determinar si un objeto JComponent es opaco (**true**) o transparente (**false**).*

Observación de apariencia visual 29.22



Los objetos JPanel son opacos de manera predeterminada.

Cuando el usuario suelta el botón del ratón, el método `mouseReleased` (línea 78) captura en las variables `x1` y `y1` la posición final del ratón e invoca al método `repaint` para dibujar la versión final del óvalo.

Cuando ejecute este programa, intente arrastrar el ratón desde el fondo de la ventana de la aplicación hacia el área del `PanelAutoContenido` para que vea que los eventos de arrastre se envían a la ventana de la aplicación, en lugar de enviarse al `PanelAutoContenido`. Después, inicie una nueva operación de arrastre en el área del `PanelAutoContenido` y arrastre el ratón hacia el fondo de la ventana de la aplicación, para que vea que los eventos de arrastre se envían al `PanelAutoContenido`, en lugar de enviarse a la ventana de la aplicación.

Observación de apariencia visual 29.23



Una operación de arrastre de ratón empieza con un evento de oprimir el botón del ratón (`mousePressed`). Todos los eventos subsecuentes de arrastre del ratón (para los cuales se hará una llamada a `mouseDragged`) se envían al componente GUI que recibió el evento original del botón oprimido del ratón.

29.14 Ventanas

Hasta este punto hemos visto muchas aplicaciones que han utilizado una subclase de `JFrame` como la GUI de la aplicación. En esta sección hablaremos sobre varias cuestiones importantes relacionadas con los objetos `JFrame`.

Un objeto `JFrame` es una *ventana* con una *barra de título* y un *borde*. La clase `JFrame` es una subclase de `java.awt.Frame` (que a su vez es una subclase de `java.awt.Window`). Como tal, `JFrame` es uno de los pocos componentes GUI de Swing que no se considera ligero. A diferencia de la mayoría de los componentes Swing, `JFrame` no está escrito completamente en Java. De hecho, si usted despliega una ventana desde un programa en Java, la ventana forma parte del conjunto de componentes GUI de la plataforma local; la ventana se verá igual que las otras ventanas que se desplieguen en esa plataforma. Cuando un programa en Java se ejecuta en una Macintosh y se despliega una ventana, la barra de título y los bordes de esa ventana se ven igual que las demás aplicaciones de Macintosh. Cuando un programa en Java se ejecuta en Microsoft Windows y se despliega una ventana, la barra de título y los bordes de esa ventana se ven igual que las otras aplicaciones de Microsoft Windows. Y, cuando un programa en Java se ejecuta en una plataforma Unix y se despliega una ventana, la barra de título y los bordes de esa ventana se ven igual que las otras aplicaciones Unix en esa plataforma.

La clase `JFrame` soporta tres operaciones cuando el usuario cierra la ventana. De manera predeterminada, una ventana se oculta (es decir, desaparece de la pantalla) cuando el usuario la cierra. Esto puede controlarse mediante el método `setDefaultCloseOperation` de `JFrame`. La interfaz `WindowConstants` (del paquete `javax.swing`) define tres constantes para usarse con este método: `DISPOSE_ON_CLOSE`, `DO NOTHING_ON_CLOSE` y `HIDE_ON_CLOSE` (la opción predeterminada). La mayoría de las plataformas permiten desplegar un número limitado de ventanas en la pantalla. Como tal, una ventana es un recurso valioso que debe regresarse al sistema cuando ya no se necesita. La clase `Window` (una superclase indirecta de `JFrame`) define el método `dispose` para este propósito. Cuando un objeto `Window` ya no es necesario en una aplicación, usted debe usar `dispose` explícitamente para desechar la ventana. Esto puede hacerse mediante una llamada explícita al método `dispose` del objeto `Window`, o llamando al método `setDefaultCloseOperation` con el argumento `WindowConstants.DISPOSE_ON_CLOSE`. Además, al terminar una aplicación se regresarán los recursos de ventanas al sistema. Al establecer la operación predeterminada de cierre en `DO NOTHING_ON_CLOSE`, usted estará indicando que determinará lo que debe hacerse cuando el usuario indique que la ventana debe cerrarse.

Observación de ingeniería de software 29.4



Las ventanas son un recurso valioso del sistema, por lo que deben regresársele cuando ya no se les necesite.

De manera predeterminada, una ventana no se despliega en la pantalla sino hasta que se llama a su método `show`. Una ventana también puede mostrarse, llamando a su método `setVisible` (heredado de la clase `java.awt.Component`), con `true` como argumento. Además, el tamaño de una ventana debe establecerse mediante una llamada al método `setSize` (heredado de la clase `java.awt.Component`). La posición de una ventana al aparecer en la pantalla se especifica con el método `setLocation` (heredado de la clase `java.awt.Component`).

Error común de programación 29.8



Ovidar llamar al método `show` o al método `setVisible` en una ventana, es un error lógico en tiempo de ejecución; la ventana no se desplegará en pantalla.

Error común de programación 29.9



Ovidar llamar al método `setSize` en una ventana, es un error lógico en tiempo de ejecución; sólo aparecerá la barra de título.

Todas las ventanas generan *eventos de ventana* cuando el usuario las manipula. Los componentes que escuchan eventos se registran para los eventos de ventana por medio del método `addWindowListener` de `Window`. La interfaz `WindowListener` (implementada por los componentes de eventos de ventana) proporciona siete métodos para manejar los eventos de ventana: `windowActivated` (se llama cuando la ventana se activa al hacer clic en ella), `windowClosed` (se llama después de cerrar la ventana), `windowClosing` (se llama cuando el usuario inicia la operación de cierre de ventana), `windowDeactivated` (se llama cuando otra ventana se activa), `windowIconified` (se llama cuando el usuario minimiza una ventana), `windowDeiconified` (se llama cuando se restaura una ventana después de ser minimizada) y `windowOpened` (se llama cuando se despliega por primera vez una ventana en la pantalla).

La mayoría de las ventanas tienen un ícono en la esquina superior izquierda o derecha, el cual permite al usuario cerrar la ventana y terminar el programa. La mayoría de las ventanas tienen también un ícono en la esquina superior izquierda de la ventana, el cual despliega un menú cuando el usuario hace clic en el ícono. Este menú por lo general contiene una opción **Cerrar** para cerrar la ventana y otras opciones para manipularla.

29.15 Uso de menús con marcos

Los *menús* son una parte integral de las GUIs. Los menús permiten al usuario realizar acciones sin “atestar” innecesariamente una interfaz gráfica de usuario con componentes GUI adicionales. En las GUIs de Swing, los menús pueden adjuntarse solamente a objetos de las clases que proporcionan el método `setJMenuBar`. Dos de esas clases son `JFrame` y `JApplet`. Las clases que se utilizan para definir menús son `JMenuBar`, `JMenuItem`, `JMenu`, `JCheckBoxMenuItem` y la clase `JRadioButtonMenuItem`.

Observación de apariencia visual 29.24



Los menús simplifican las GUIs, al reducir el número de componentes que ve el usuario.

La clase `JMenuBar` (una subclase de `JComponent`) contiene los métodos necesarios para administrar una *barra de menús*, la cual es un contenedor de menús.

La clase `JMenuItem` (una subclase de `javax.swing.AbstractButton`) contiene los métodos necesarios para administrar *elementos de menú*. Un elemento de menú es un componente GUI que se encuentra en un menú que, al ser seleccionado, hace que se realice una acción. Un elemento de menú puede usarse para iniciar una acción, o puede ser un *submenú* que proporcione más elementos de menú que pueda seleccionar el usuario. Los submenús son útiles para agrupar en un menú varios elementos de menú relacionados.

La clase `JMenu` (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar *menús*. Los menús contienen elementos de menú y se agregan a las barras de menús o a otros menús como submenús. Al hacer clic en un menú, éste se expande para mostrar su lista de elementos. Al hacer clic en uno de los elementos del menú se genera un evento de acción.

La clase `JCheckBoxMenuItem` (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar elementos de menú que pueden activarse o desactivarse. Cuando se selecciona un objeto `JCheckBoxMenuItem`, aparece una marca de verificación a la izquierda de ese elemento de menú. Cuando el objeto `JCheckBoxMenuItem` se selecciona nuevamente, se quita la marca de verificación que está a la izquierda del elemento de menú.

La clase `JRadioButtonMenuItem` (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar elementos de menú que pueden activarse o desactivarse de igual forma que los objetos `JCheckBoxMenuItem`. Cuando se mantienen varios objetos `JRadioButtonMenuItem` como parte de un grupo de botones (`ButtonGroup`), sólo puede seleccionarse un elemento del grupo a la vez. Cuando se selecciona un objeto `JRadioButtonMenuItem`, aparece un círculo relleno a la izquierda del ele-

mento de menú. Cuando se selecciona otro objeto **JRadioButtonMenuItem**, se quita el círculo relleno que está a la izquierda del elemento de menú previamente seleccionado.

La aplicación de la figura 29.22 muestra el uso de varios tipos de elementos de menú. El programa también muestra cómo especificar caracteres especiales (conocidos como mnemónicos) que pueden proporcionar un acceso rápido a un menú o elemento de menú desde el teclado. Los mnemónicos pueden utilizarse con objetos de cualquier clase que sea subclase de **java.swing.AbstractButton**.

```
1 // Figura 29.22: PruebaMenu.java
2 // Demostración del uso de menús
3 import javax.swing.*;
4 import java.awt.event.*;
5 import java.awt.*;
6
7 public class PruebaMenu extends JFrame {
8     private Color valoresColores[] =
9         { Color.black, Color.blue, Color.red, Color.green };
10    private JRadioButtonMenuItem elementosColores[], tiposDeLetra[];
11    private JCheckBoxMenuItem elementosEstilo[];
12    private JLabel pantalla;
13    private ButtonGroup grupoTiposDeLetra, grupoColores;
14    private int estilo;
15
16    public PruebaMenu()
17    {
18        super( "Uso de objetos JMenu" );
19
20        JMenuBar barra = new JMenuBar(); // crea la barra de menús
21        setJMenuBar( barra ); // establece la barra de menús para el objeto
22        // JFrame
23
24        // crea el menú Archivo y el elemento de menú Salir
25        JMenu menuArchivo = new JMenu( "Archivo" );
26        menuArchivo.setMnemonic( 'A' );
27        JMenuItem elementoAcercaDe = new JMenuItem( "Acerca de..." );
28        elementoAcercaDe.setMnemonic( 'c' );
29        elementoAcercaDe.addActionListener(
30            new ActionListener() {
31                public void actionPerformed( ActionEvent e )
32                {
33                    JOptionPane.showMessageDialog( PruebaMenu.this,
34                        "Este es un ejemplo\nde del uso de menus",
35                        "Acerca de", JOptionPane.PLAIN_MESSAGE );
36                } // fin del método actionPerformed
37            } // fin de la clase interna anónima
38        ); // fin de addActionListener
39        menuArchivo.add( elementoAcercaDe );
40
41        JMenuItem elementoSalir = new JMenuItem( "Salir" );
42        elementoSalir.setMnemonic( 'S' );
43        elementoSalir.addActionListener(
44            new ActionListener() {
45                public void actionPerformed( ActionEvent e )
46                {
47                    System.exit( 0 );
48                }
49            }
50        );
51
52        // crea el menú Ayuda y el elemento de menú Salir
53        JMenu menuAyuda = new JMenu( "Ayuda" );
54        menuAyuda.setMnemonic( 'H' );
55        JMenuItem elementoAyuda = new JMenuItem( "Ayuda" );
56        elementoAyuda.setMnemonic( 'y' );
57        elementoAyuda.addActionListener(
58            new ActionListener() {
59                public void actionPerformed( ActionEvent e )
60                {
61                    JOptionPane.showMessageDialog( PruebaMenu.this,
62                        "Este es un ejemplo\nde del uso de menus",
63                        "Ayuda", JOptionPane.PLAIN_MESSAGE );
64                }
65            }
66        );
67        menuAyuda.add( elementoAyuda );
68
69        // crea el menú Herramientas y el elemento de menú Salir
70        JMenu menuHerramientas = new JMenu( "Herramientas" );
71        menuHerramientas.setMnemonic( 'T' );
72        JMenuItem elementoHerramientas = new JMenuItem( "Herramientas" );
73        elementoHerramientas.setMnemonic( 'h' );
74        elementoHerramientas.addActionListener(
75            new ActionListener() {
76                public void actionPerformed( ActionEvent e )
77                {
78                    JOptionPane.showMessageDialog( PruebaMenu.this,
79                        "Este es un ejemplo\nde del uso de menus",
80                        "Herramientas", JOptionPane.PLAIN_MESSAGE );
81                }
82            }
83        );
84        menuHerramientas.add( elementoHerramientas );
85
86        // crea el menú Edición y el elemento de menú Salir
87        JMenu menuEdicion = new JMenu( "Edición" );
88        menuEdicion.setMnemonic( 'E' );
89        JMenuItem elementoEdicion = new JMenuItem( "Edición" );
90        elementoEdicion.setMnemonic( 'e' );
91        elementoEdicion.addActionListener(
92            new ActionListener() {
93                public void actionPerformed( ActionEvent e )
94                {
95                    JOptionPane.showMessageDialog( PruebaMenu.this,
96                        "Este es un ejemplo\nde del uso de menus",
97                        "Edición", JOptionPane.PLAIN_MESSAGE );
98                }
99            }
100       );
101      menuEdicion.add( elementoEdicion );
102
103      // crea el menú Ver y el elemento de menú Salir
104      JMenu menuVer = new JMenu( "Ver" );
105      menuVer.setMnemonic( 'V' );
106      JMenuItem elementoVer = new JMenuItem( "Ver" );
107      elementoVer.setMnemonic( 'v' );
108      elementoVer.addActionListener(
109          new ActionListener() {
110              public void actionPerformed( ActionEvent e )
111              {
112                  JOptionPane.showMessageDialog( PruebaMenu.this,
113                      "Este es un ejemplo\nde del uso de menus",
114                      "Ver", JOptionPane.PLAIN_MESSAGE );
115              }
116          }
117      );
118      menuVer.add( elementoVer );
119
120      // crea el menú Ayuda y el elemento de menú Salir
121      JMenu menuAyuda2 = new JMenu( "Ayuda" );
122      menuAyuda2.setMnemonic( 'H' );
123      JMenuItem elementoAyuda2 = new JMenuItem( "Ayuda" );
124      elementoAyuda2.setMnemonic( 'y' );
125      elementoAyuda2.addActionListener(
126          new ActionListener() {
127              public void actionPerformed( ActionEvent e )
128              {
129                  JOptionPane.showMessageDialog( PruebaMenu.this,
130                      "Este es un ejemplo\nde del uso de menus",
131                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
132              }
133          }
134      );
135      menuAyuda2.add( elementoAyuda2 );
136
137      // crea el menú Herramientas y el elemento de menú Salir
138      JMenu menuHerramientas2 = new JMenu( "Herramientas" );
139      menuHerramientas2.setMnemonic( 'T' );
140      JMenuItem elementoHerramientas2 = new JMenuItem( "Herramientas" );
141      elementoHerramientas2.setMnemonic( 'h' );
142      elementoHerramientas2.addActionListener(
143          new ActionListener() {
144              public void actionPerformed( ActionEvent e )
145              {
146                  JOptionPane.showMessageDialog( PruebaMenu.this,
147                      "Este es un ejemplo\nde del uso de menus",
148                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
149              }
150          }
151      );
152      menuHerramientas2.add( elementoHerramientas2 );
153
154      // crea el menú Edición y el elemento de menú Salir
155      JMenu menuEdicion2 = new JMenu( "Edición" );
156      menuEdicion2.setMnemonic( 'E' );
157      JMenuItem elementoEdicion2 = new JMenuItem( "Edición" );
158      elementoEdicion2.setMnemonic( 'e' );
159      elementoEdicion2.addActionListener(
160          new ActionListener() {
161              public void actionPerformed( ActionEvent e )
162              {
163                  JOptionPane.showMessageDialog( PruebaMenu.this,
164                      "Este es un ejemplo\nde del uso de menus",
165                      "Edición", JOptionPane.PLAIN_MESSAGE );
166              }
167          }
168      );
169      menuEdicion2.add( elementoEdicion2 );
170
171      // crea el menú Ver y el elemento de menú Salir
172      JMenu menuVer2 = new JMenu( "Ver" );
173      menuVer2.setMnemonic( 'V' );
174      JMenuItem elementoVer2 = new JMenuItem( "Ver" );
175      elementoVer2.setMnemonic( 'v' );
176      elementoVer2.addActionListener(
177          new ActionListener() {
178              public void actionPerformed( ActionEvent e )
179              {
180                  JOptionPane.showMessageDialog( PruebaMenu.this,
181                      "Este es un ejemplo\nde del uso de menus",
182                      "Ver", JOptionPane.PLAIN_MESSAGE );
183              }
184          }
185      );
186      menuVer2.add( elementoVer2 );
187
188      // crea el menú Ayuda y el elemento de menú Salir
189      JMenu menuAyuda3 = new JMenu( "Ayuda" );
190      menuAyuda3.setMnemonic( 'H' );
191      JMenuItem elementoAyuda3 = new JMenuItem( "Ayuda" );
192      elementoAyuda3.setMnemonic( 'y' );
193      elementoAyuda3.addActionListener(
194          new ActionListener() {
195              public void actionPerformed( ActionEvent e )
196              {
197                  JOptionPane.showMessageDialog( PruebaMenu.this,
198                      "Este es un ejemplo\nde del uso de menus",
199                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
200              }
201          }
202      );
203      menuAyuda3.add( elementoAyuda3 );
204
205      // crea el menú Herramientas y el elemento de menú Salir
206      JMenu menuHerramientas3 = new JMenu( "Herramientas" );
207      menuHerramientas3.setMnemonic( 'T' );
208      JMenuItem elementoHerramientas3 = new JMenuItem( "Herramientas" );
209      elementoHerramientas3.setMnemonic( 'h' );
210      elementoHerramientas3.addActionListener(
211          new ActionListener() {
212              public void actionPerformed( ActionEvent e )
213              {
214                  JOptionPane.showMessageDialog( PruebaMenu.this,
215                      "Este es un ejemplo\nde del uso de menus",
216                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
217              }
218          }
219      );
220      menuHerramientas3.add( elementoHerramientas3 );
221
222      // crea el menú Edición y el elemento de menú Salir
223      JMenu menuEdicion3 = new JMenu( "Edición" );
224      menuEdicion3.setMnemonic( 'E' );
225      JMenuItem elementoEdicion3 = new JMenuItem( "Edición" );
226      elementoEdicion3.setMnemonic( 'e' );
227      elementoEdicion3.addActionListener(
228          new ActionListener() {
229              public void actionPerformed( ActionEvent e )
230              {
231                  JOptionPane.showMessageDialog( PruebaMenu.this,
232                      "Este es un ejemplo\nde del uso de menus",
233                      "Edición", JOptionPane.PLAIN_MESSAGE );
234              }
235          }
236      );
237      menuEdicion3.add( elementoEdicion3 );
238
239      // crea el menú Ver y el elemento de menú Salir
240      JMenu menuVer3 = new JMenu( "Ver" );
241      menuVer3.setMnemonic( 'V' );
242      JMenuItem elementoVer3 = new JMenuItem( "Ver" );
243      elementoVer3.setMnemonic( 'v' );
244      elementoVer3.addActionListener(
245          new ActionListener() {
246              public void actionPerformed( ActionEvent e )
247              {
248                  JOptionPane.showMessageDialog( PruebaMenu.this,
249                      "Este es un ejemplo\nde del uso de menus",
250                      "Ver", JOptionPane.PLAIN_MESSAGE );
251              }
252          }
253      );
254      menuVer3.add( elementoVer3 );
255
256      // crea el menú Ayuda y el elemento de menú Salir
257      JMenu menuAyuda4 = new JMenu( "Ayuda" );
258      menuAyuda4.setMnemonic( 'H' );
259      JMenuItem elementoAyuda4 = new JMenuItem( "Ayuda" );
260      elementoAyuda4.setMnemonic( 'y' );
261      elementoAyuda4.addActionListener(
262          new ActionListener() {
263              public void actionPerformed( ActionEvent e )
264              {
265                  JOptionPane.showMessageDialog( PruebaMenu.this,
266                      "Este es un ejemplo\nde del uso de menus",
267                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
268              }
269          }
270      );
271      menuAyuda4.add( elementoAyuda4 );
272
273      // crea el menú Herramientas y el elemento de menú Salir
274      JMenu menuHerramientas4 = new JMenu( "Herramientas" );
275      menuHerramientas4.setMnemonic( 'T' );
276      JMenuItem elementoHerramientas4 = new JMenuItem( "Herramientas" );
277      elementoHerramientas4.setMnemonic( 'h' );
278      elementoHerramientas4.addActionListener(
279          new ActionListener() {
280              public void actionPerformed( ActionEvent e )
281              {
282                  JOptionPane.showMessageDialog( PruebaMenu.this,
283                      "Este es un ejemplo\nde del uso de menus",
284                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
285              }
286          }
287      );
288      menuHerramientas4.add( elementoHerramientas4 );
289
290      // crea el menú Edición y el elemento de menú Salir
291      JMenu menuEdicion4 = new JMenu( "Edición" );
292      menuEdicion4.setMnemonic( 'E' );
293      JMenuItem elementoEdicion4 = new JMenuItem( "Edición" );
294      elementoEdicion4.setMnemonic( 'e' );
295      elementoEdicion4.addActionListener(
296          new ActionListener() {
297              public void actionPerformed( ActionEvent e )
298              {
299                  JOptionPane.showMessageDialog( PruebaMenu.this,
300                      "Este es un ejemplo\nde del uso de menus",
301                      "Edición", JOptionPane.PLAIN_MESSAGE );
302              }
303          }
304      );
305      menuEdicion4.add( elementoEdicion4 );
306
307      // crea el menú Ver y el elemento de menú Salir
308      JMenu menuVer4 = new JMenu( "Ver" );
309      menuVer4.setMnemonic( 'V' );
310      JMenuItem elementoVer4 = new JMenuItem( "Ver" );
311      elementoVer4.setMnemonic( 'v' );
312      elementoVer4.addActionListener(
313          new ActionListener() {
314              public void actionPerformed( ActionEvent e )
315              {
316                  JOptionPane.showMessageDialog( PruebaMenu.this,
317                      "Este es un ejemplo\nde del uso de menus",
318                      "Ver", JOptionPane.PLAIN_MESSAGE );
319              }
320          }
321      );
322      menuVer4.add( elementoVer4 );
323
324      // crea el menú Ayuda y el elemento de menú Salir
325      JMenu menuAyuda5 = new JMenu( "Ayuda" );
326      menuAyuda5.setMnemonic( 'H' );
327      JMenuItem elementoAyuda5 = new JMenuItem( "Ayuda" );
328      elementoAyuda5.setMnemonic( 'y' );
329      elementoAyuda5.addActionListener(
330          new ActionListener() {
331              public void actionPerformed( ActionEvent e )
332              {
333                  JOptionPane.showMessageDialog( PruebaMenu.this,
334                      "Este es un ejemplo\nde del uso de menus",
335                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
336              }
337          }
338      );
339      menuAyuda5.add( elementoAyuda5 );
340
341      // crea el menú Herramientas y el elemento de menú Salir
342      JMenu menuHerramientas5 = new JMenu( "Herramientas" );
343      menuHerramientas5.setMnemonic( 'T' );
344      JMenuItem elementoHerramientas5 = new JMenuItem( "Herramientas" );
345      elementoHerramientas5.setMnemonic( 'h' );
346      elementoHerramientas5.addActionListener(
347          new ActionListener() {
348              public void actionPerformed( ActionEvent e )
349              {
350                  JOptionPane.showMessageDialog( PruebaMenu.this,
351                      "Este es un ejemplo\nde del uso de menus",
352                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
353              }
354          }
355      );
356      menuHerramientas5.add( elementoHerramientas5 );
357
358      // crea el menú Edición y el elemento de menú Salir
359      JMenu menuEdicion5 = new JMenu( "Edición" );
360      menuEdicion5.setMnemonic( 'E' );
361      JMenuItem elementoEdicion5 = new JMenuItem( "Edición" );
362      elementoEdicion5.setMnemonic( 'e' );
363      elementoEdicion5.addActionListener(
364          new ActionListener() {
365              public void actionPerformed( ActionEvent e )
366              {
367                  JOptionPane.showMessageDialog( PruebaMenu.this,
368                      "Este es un ejemplo\nde del uso de menus",
369                      "Edición", JOptionPane.PLAIN_MESSAGE );
370              }
371          }
372      );
373      menuEdicion5.add( elementoEdicion5 );
374
375      // crea el menú Ver y el elemento de menú Salir
376      JMenu menuVer5 = new JMenu( "Ver" );
377      menuVer5.setMnemonic( 'V' );
378      JMenuItem elementoVer5 = new JMenuItem( "Ver" );
379      elementoVer5.setMnemonic( 'v' );
380      elementoVer5.addActionListener(
381          new ActionListener() {
382              public void actionPerformed( ActionEvent e )
383              {
384                  JOptionPane.showMessageDialog( PruebaMenu.this,
385                      "Este es un ejemplo\nde del uso de menus",
386                      "Ver", JOptionPane.PLAIN_MESSAGE );
387              }
388          }
389      );
390      menuVer5.add( elementoVer5 );
391
392      // crea el menú Ayuda y el elemento de menú Salir
393      JMenu menuAyuda6 = new JMenu( "Ayuda" );
394      menuAyuda6.setMnemonic( 'H' );
395      JMenuItem elementoAyuda6 = new JMenuItem( "Ayuda" );
396      elementoAyuda6.setMnemonic( 'y' );
397      elementoAyuda6.addActionListener(
398          new ActionListener() {
399              public void actionPerformed( ActionEvent e )
400              {
401                  JOptionPane.showMessageDialog( PruebaMenu.this,
402                      "Este es un ejemplo\nde del uso de menus",
403                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
404              }
405          }
406      );
407      menuAyuda6.add( elementoAyuda6 );
408
409      // crea el menú Herramientas y el elemento de menú Salir
410      JMenu menuHerramientas6 = new JMenu( "Herramientas" );
411      menuHerramientas6.setMnemonic( 'T' );
412      JMenuItem elementoHerramientas6 = new JMenuItem( "Herramientas" );
413      elementoHerramientas6.setMnemonic( 'h' );
414      elementoHerramientas6.addActionListener(
415          new ActionListener() {
416              public void actionPerformed( ActionEvent e )
417              {
418                  JOptionPane.showMessageDialog( PruebaMenu.this,
419                      "Este es un ejemplo\nde del uso de menus",
420                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
421              }
422          }
423      );
424      menuHerramientas6.add( elementoHerramientas6 );
425
426      // crea el menú Edición y el elemento de menú Salir
427      JMenu menuEdicion6 = new JMenu( "Edición" );
428      menuEdicion6.setMnemonic( 'E' );
429      JMenuItem elementoEdicion6 = new JMenuItem( "Edición" );
430      elementoEdicion6.setMnemonic( 'e' );
431      elementoEdicion6.addActionListener(
432          new ActionListener() {
433              public void actionPerformed( ActionEvent e )
434              {
435                  JOptionPane.showMessageDialog( PruebaMenu.this,
436                      "Este es un ejemplo\nde del uso de menus",
437                      "Edición", JOptionPane.PLAIN_MESSAGE );
438              }
439          }
440      );
441      menuEdicion6.add( elementoEdicion6 );
442
443      // crea el menú Ver y el elemento de menú Salir
444      JMenu menuVer6 = new JMenu( "Ver" );
445      menuVer6.setMnemonic( 'V' );
446      JMenuItem elementoVer6 = new JMenuItem( "Ver" );
447      elementoVer6.setMnemonic( 'v' );
448      elementoVer6.addActionListener(
449          new ActionListener() {
450              public void actionPerformed( ActionEvent e )
451              {
452                  JOptionPane.showMessageDialog( PruebaMenu.this,
453                      "Este es un ejemplo\nde del uso de menus",
454                      "Ver", JOptionPane.PLAIN_MESSAGE );
455              }
456          }
457      );
458      menuVer6.add( elementoVer6 );
459
460      // crea el menú Ayuda y el elemento de menú Salir
461      JMenu menuAyuda7 = new JMenu( "Ayuda" );
462      menuAyuda7.setMnemonic( 'H' );
463      JMenuItem elementoAyuda7 = new JMenuItem( "Ayuda" );
464      elementoAyuda7.setMnemonic( 'y' );
465      elementoAyuda7.addActionListener(
466          new ActionListener() {
467              public void actionPerformed( ActionEvent e )
468              {
469                  JOptionPane.showMessageDialog( PruebaMenu.this,
470                      "Este es un ejemplo\nde del uso de menus",
471                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
472              }
473          }
474      );
475      menuAyuda7.add( elementoAyuda7 );
476
477      // crea el menú Herramientas y el elemento de menú Salir
478      JMenu menuHerramientas7 = new JMenu( "Herramientas" );
479      menuHerramientas7.setMnemonic( 'T' );
480      JMenuItem elementoHerramientas7 = new JMenuItem( "Herramientas" );
481      elementoHerramientas7.setMnemonic( 'h' );
482      elementoHerramientas7.addActionListener(
483          new ActionListener() {
484              public void actionPerformed( ActionEvent e )
485              {
486                  JOptionPane.showMessageDialog( PruebaMenu.this,
487                      "Este es un ejemplo\nde del uso de menus",
488                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
489              }
490          }
491      );
492      menuHerramientas7.add( elementoHerramientas7 );
493
494      // crea el menú Edición y el elemento de menú Salir
495      JMenu menuEdicion7 = new JMenu( "Edición" );
496      menuEdicion7.setMnemonic( 'E' );
497      JMenuItem elementoEdicion7 = new JMenuItem( "Edición" );
498      elementoEdicion7.setMnemonic( 'e' );
499      elementoEdicion7.addActionListener(
500          new ActionListener() {
501              public void actionPerformed( ActionEvent e )
502              {
503                  JOptionPane.showMessageDialog( PruebaMenu.this,
504                      "Este es un ejemplo\nde del uso de menus",
505                      "Edición", JOptionPane.PLAIN_MESSAGE );
506              }
507          }
508      );
509      menuEdicion7.add( elementoEdicion7 );
510
511      // crea el menú Ver y el elemento de menú Salir
512      JMenu menuVer7 = new JMenu( "Ver" );
513      menuVer7.setMnemonic( 'V' );
514      JMenuItem elementoVer7 = new JMenuItem( "Ver" );
515      elementoVer7.setMnemonic( 'v' );
516      elementoVer7.addActionListener(
517          new ActionListener() {
518              public void actionPerformed( ActionEvent e )
519              {
520                  JOptionPane.showMessageDialog( PruebaMenu.this,
521                      "Este es un ejemplo\nde del uso de menus",
522                      "Ver", JOptionPane.PLAIN_MESSAGE );
523              }
524          }
525      );
526      menuVer7.add( elementoVer7 );
527
528      // crea el menú Ayuda y el elemento de menú Salir
529      JMenu menuAyuda8 = new JMenu( "Ayuda" );
530      menuAyuda8.setMnemonic( 'H' );
531      JMenuItem elementoAyuda8 = new JMenuItem( "Ayuda" );
532      elementoAyuda8.setMnemonic( 'y' );
533      elementoAyuda8.addActionListener(
534          new ActionListener() {
535              public void actionPerformed( ActionEvent e )
536              {
537                  JOptionPane.showMessageDialog( PruebaMenu.this,
538                      "Este es un ejemplo\nde del uso de menus",
539                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
540              }
541          }
542      );
543      menuAyuda8.add( elementoAyuda8 );
544
545      // crea el menú Herramientas y el elemento de menú Salir
546      JMenu menuHerramientas8 = new JMenu( "Herramientas" );
547      menuHerramientas8.setMnemonic( 'T' );
548      JMenuItem elementoHerramientas8 = new JMenuItem( "Herramientas" );
549      elementoHerramientas8.setMnemonic( 'h' );
550      elementoHerramientas8.addActionListener(
551          new ActionListener() {
552              public void actionPerformed( ActionEvent e )
553              {
554                  JOptionPane.showMessageDialog( PruebaMenu.this,
555                      "Este es un ejemplo\nde del uso de menus",
556                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
557              }
558          }
559      );
560      menuHerramientas8.add( elementoHerramientas8 );
561
562      // crea el menú Edición y el elemento de menú Salir
563      JMenu menuEdicion8 = new JMenu( "Edición" );
564      menuEdicion8.setMnemonic( 'E' );
565      JMenuItem elementoEdicion8 = new JMenuItem( "Edición" );
566      elementoEdicion8.setMnemonic( 'e' );
567      elementoEdicion8.addActionListener(
568          new ActionListener() {
569              public void actionPerformed( ActionEvent e )
570              {
571                  JOptionPane.showMessageDialog( PruebaMenu.this,
572                      "Este es un ejemplo\nde del uso de menus",
573                      "Edición", JOptionPane.PLAIN_MESSAGE );
574              }
575          }
576      );
577      menuEdicion8.add( elementoEdicion8 );
578
579      // crea el menú Ver y el elemento de menú Salir
580      JMenu menuVer8 = new JMenu( "Ver" );
581      menuVer8.setMnemonic( 'V' );
582      JMenuItem elementoVer8 = new JMenuItem( "Ver" );
583      elementoVer8.setMnemonic( 'v' );
584      elementoVer8.addActionListener(
585          new ActionListener() {
586              public void actionPerformed( ActionEvent e )
587              {
588                  JOptionPane.showMessageDialog( PruebaMenu.this,
589                      "Este es un ejemplo\nde del uso de menus",
590                      "Ver", JOptionPane.PLAIN_MESSAGE );
591              }
592          }
593      );
594      menuVer8.add( elementoVer8 );
595
596      // crea el menú Ayuda y el elemento de menú Salir
597      JMenu menuAyuda9 = new JMenu( "Ayuda" );
598      menuAyuda9.setMnemonic( 'H' );
599      JMenuItem elementoAyuda9 = new JMenuItem( "Ayuda" );
600      elementoAyuda9.setMnemonic( 'y' );
601      elementoAyuda9.addActionListener(
602          new ActionListener() {
603              public void actionPerformed( ActionEvent e )
604              {
605                  JOptionPane.showMessageDialog( PruebaMenu.this,
606                      "Este es un ejemplo\nde del uso de menus",
607                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
608              }
609          }
610      );
611      menuAyuda9.add( elementoAyuda9 );
612
613      // crea el menú Herramientas y el elemento de menú Salir
614      JMenu menuHerramientas9 = new JMenu( "Herramientas" );
615      menuHerramientas9.setMnemonic( 'T' );
616      JMenuItem elementoHerramientas9 = new JMenuItem( "Herramientas" );
617      elementoHerramientas9.setMnemonic( 'h' );
618      elementoHerramientas9.addActionListener(
619          new ActionListener() {
620              public void actionPerformed( ActionEvent e )
621              {
622                  JOptionPane.showMessageDialog( PruebaMenu.this,
623                      "Este es un ejemplo\nde del uso de menus",
624                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
625              }
626          }
627      );
628      menuHerramientas9.add( elementoHerramientas9 );
629
630      // crea el menú Edición y el elemento de menú Salir
631      JMenu menuEdicion9 = new JMenu( "Edición" );
632      menuEdicion9.setMnemonic( 'E' );
633      JMenuItem elementoEdicion9 = new JMenuItem( "Edición" );
634      elementoEdicion9.setMnemonic( 'e' );
635      elementoEdicion9.addActionListener(
636          new ActionListener() {
637              public void actionPerformed( ActionEvent e )
638              {
639                  JOptionPane.showMessageDialog( PruebaMenu.this,
640                      "Este es un ejemplo\nde del uso de menus",
641                      "Edición", JOptionPane.PLAIN_MESSAGE );
642              }
643          }
644      );
645      menuEdicion9.add( elementoEdicion9 );
646
647      // crea el menú Ver y el elemento de menú Salir
648      JMenu menuVer9 = new JMenu( "Ver" );
649      menuVer9.setMnemonic( 'V' );
650      JMenuItem elementoVer9 = new JMenuItem( "Ver" );
651      elementoVer9.setMnemonic( 'v' );
652      elementoVer9.addActionListener(
653          new ActionListener() {
654              public void actionPerformed( ActionEvent e )
655              {
656                  JOptionPane.showMessageDialog( PruebaMenu.this,
657                      "Este es un ejemplo\nde del uso de menus",
658                      "Ver", JOptionPane.PLAIN_MESSAGE );
659              }
660          }
661      );
662      menuVer9.add( elementoVer9 );
663
664      // crea el menú Ayuda y el elemento de menú Salir
665      JMenu menuAyuda10 = new JMenu( "Ayuda" );
666      menuAyuda10.setMnemonic( 'H' );
667      JMenuItem elementoAyuda10 = new JMenuItem( "Ayuda" );
668      elementoAyuda10.setMnemonic( 'y' );
669      elementoAyuda10.addActionListener(
670          new ActionListener() {
671              public void actionPerformed( ActionEvent e )
672              {
673                  JOptionPane.showMessageDialog( PruebaMenu.this,
674                      "Este es un ejemplo\nde del uso de menus",
675                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
676              }
677          }
678      );
679      menuAyuda10.add( elementoAyuda10 );
680
681      // crea el menú Herramientas y el elemento de menú Salir
682      JMenu menuHerramientas10 = new JMenu( "Herramientas" );
683      menuHerramientas10.setMnemonic( 'T' );
684      JMenuItem elementoHerramientas10 = new JMenuItem( "Herramientas" );
685      elementoHerramientas10.setMnemonic( 'h' );
686      elementoHerramientas10.addActionListener(
687          new ActionListener() {
688              public void actionPerformed( ActionEvent e )
689              {
690                  JOptionPane.showMessageDialog( PruebaMenu.this,
691                      "Este es un ejemplo\nde del uso de menus",
692                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
693              }
694          }
695      );
696      menuHerramientas10.add( elementoHerramientas10 );
697
698      // crea el menú Edición y el elemento de menú Salir
699      JMenu menuEdicion10 = new JMenu( "Edición" );
700      menuEdicion10.setMnemonic( 'E' );
701      JMenuItem elementoEdicion10 = new JMenuItem( "Edición" );
702      elementoEdicion10.setMnemonic( 'e' );
703      elementoEdicion10.addActionListener(
704          new ActionListener() {
705              public void actionPerformed( ActionEvent e )
706              {
707                  JOptionPane.showMessageDialog( PruebaMenu.this,
708                      "Este es un ejemplo\nde del uso de menus",
709                      "Edición", JOptionPane.PLAIN_MESSAGE );
710              }
711          }
712      );
713      menuEdicion10.add( elementoEdicion10 );
714
715      // crea el menú Ver y el elemento de menú Salir
716      JMenu menuVer10 = new JMenu( "Ver" );
717      menuVer10.setMnemonic( 'V' );
718      JMenuItem elementoVer10 = new JMenuItem( "Ver" );
719      elementoVer10.setMnemonic( 'v' );
720      elementoVer10.addActionListener(
721          new ActionListener() {
722              public void actionPerformed( ActionEvent e )
723              {
724                  JOptionPane.showMessageDialog( PruebaMenu.this,
725                      "Este es un ejemplo\nde del uso de menus",
726                      "Ver", JOptionPane.PLAIN_MESSAGE );
727              }
728          }
729      );
730      menuVer10.add( elementoVer10 );
731
732      // crea el menú Ayuda y el elemento de menú Salir
733      JMenu menuAyuda11 = new JMenu( "Ayuda" );
734      menuAyuda11.setMnemonic( 'H' );
735      JMenuItem elementoAyuda11 = new JMenuItem( "Ayuda" );
736      elementoAyuda11.setMnemonic( 'y' );
737      elementoAyuda11.addActionListener(
738          new ActionListener() {
739              public void actionPerformed( ActionEvent e )
740              {
741                  JOptionPane.showMessageDialog( PruebaMenu.this,
742                      "Este es un ejemplo\nde del uso de menus",
743                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
744              }
745          }
746      );
747      menuAyuda11.add( elementoAyuda11 );
748
749      // crea el menú Herramientas y el elemento de menú Salir
750      JMenu menuHerramientas11 = new JMenu( "Herramientas" );
751      menuHerramientas11.setMnemonic( 'T' );
752      JMenuItem elementoHerramientas11 = new JMenuItem( "Herramientas" );
753      elementoHerramientas11.setMnemonic( 'h' );
754      elementoHerramientas11.addActionListener(
755          new ActionListener() {
756              public void actionPerformed( ActionEvent e )
757              {
758                  JOptionPane.showMessageDialog( PruebaMenu.this,
759                      "Este es un ejemplo\nde del uso de menus",
760                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
761              }
762          }
763      );
764      menuHerramientas11.add( elementoHerramientas11 );
765
766      // crea el menú Edición y el elemento de menú Salir
767      JMenu menuEdicion11 = new JMenu( "Edición" );
768      menuEdicion11.setMnemonic( 'E' );
769      JMenuItem elementoEdicion11 = new JMenuItem( "Edición" );
770      elementoEdicion11.setMnemonic( 'e' );
771      elementoEdicion11.addActionListener(
772          new ActionListener() {
773              public void actionPerformed( ActionEvent e )
774              {
775                  JOptionPane.showMessageDialog( PruebaMenu.this,
776                      "Este es un ejemplo\nde del uso de menus",
777                      "Edición", JOptionPane.PLAIN_MESSAGE );
778              }
779          }
780      );
781      menuEdicion11.add( elementoEdicion11 );
782
783      // crea el menú Ver y el elemento de menú Salir
784      JMenu menuVer11 = new JMenu( "Ver" );
785      menuVer11.setMnemonic( 'V' );
786      JMenuItem elementoVer11 = new JMenuItem( "Ver" );
787      elementoVer11.setMnemonic( 'v' );
788      elementoVer11.addActionListener(
789          new ActionListener() {
790              public void actionPerformed( ActionEvent e )
791              {
792                  JOptionPane.showMessageDialog( PruebaMenu.this,
793                      "Este es un ejemplo\nde del uso de menus",
794                      "Ver", JOptionPane.PLAIN_MESSAGE );
795              }
796          }
797      );
798      menuVer11.add( elementoVer11 );
799
800      // crea el menú Ayuda y el elemento de menú Salir
801      JMenu menuAyuda12 = new JMenu( "Ayuda" );
802      menuAyuda12.setMnemonic( 'H' );
803      JMenuItem elementoAyuda12 = new JMenuItem( "Ayuda" );
804      elementoAyuda12.setMnemonic( 'y' );
805      elementoAyuda12.addActionListener(
806          new ActionListener() {
807              public void actionPerformed( ActionEvent e )
808              {
809                  JOptionPane.showMessageDialog( PruebaMenu.this,
810                      "Este es un ejemplo\nde del uso de menus",
811                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
812              }
813          }
814      );
815      menuAyuda12.add( elementoAyuda12 );
816
817      // crea el menú Herramientas y el elemento de menú Salir
818      JMenu menuHerramientas12 = new JMenu( "Herramientas" );
819      menuHerramientas12.setMnemonic( 'T' );
820      JMenuItem elementoHerramientas12 = new JMenuItem( "Herramientas" );
821      elementoHerramientas12.setMnemonic( 'h' );
822      elementoHerramientas12.addActionListener(
823          new ActionListener() {
824              public void actionPerformed( ActionEvent e )
825              {
826                  JOptionPane.showMessageDialog( PruebaMenu.this,
827                      "Este es un ejemplo\nde del uso de menus",
828                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
829              }
830          }
831      );
832      menuHerramientas12.add( elementoHerramientas12 );
833
834      // crea el menú Edición y el elemento de menú Salir
835      JMenu menuEdicion12 = new JMenu( "Edición" );
836      menuEdicion12.setMnemonic( 'E' );
837      JMenuItem elementoEdicion12 = new JMenuItem( "Edición" );
838      elementoEdicion12.setMnemonic( 'e' );
839      elementoEdicion12.addActionListener(
840          new ActionListener() {
841              public void actionPerformed( ActionEvent e )
842              {
843                  JOptionPane.showMessageDialog( PruebaMenu.this,
844                      "Este es un ejemplo\nde del uso de menus",
845                      "Edición", JOptionPane.PLAIN_MESSAGE );
846              }
847          }
848      );
849      menuEdicion12.add( elementoEdicion12 );
850
851      // crea el menú Ver y el elemento de menú Salir
852      JMenu menuVer12 = new JMenu( "Ver" );
853      menuVer12.setMnemonic( 'V' );
854      JMenuItem elementoVer12 = new JMenuItem( "Ver" );
855      elementoVer12.setMnemonic( 'v' );
856      elementoVer12.addActionListener(
857          new ActionListener() {
858              public void actionPerformed( ActionEvent e )
859              {
860                  JOptionPane.showMessageDialog( PruebaMenu.this,
861                      "Este es un ejemplo\nde del uso de menus",
862                      "Ver", JOptionPane.PLAIN_MESSAGE );
863              }
864          }
865      );
866      menuVer12.add( elementoVer12 );
867
868      // crea el menú Ayuda y el elemento de menú Salir
869      JMenu menuAyuda13 = new JMenu( "Ayuda" );
870      menuAyuda13.setMnemonic( 'H' );
871      JMenuItem elementoAyuda13 = new JMenuItem( "Ayuda" );
872      elementoAyuda13.setMnemonic( 'y' );
873      elementoAyuda13.addActionListener(
874          new ActionListener() {
875              public void actionPerformed( ActionEvent e )
876              {
877                  JOptionPane.showMessageDialog( PruebaMenu.this,
878                      "Este es un ejemplo\nde del uso de menus",
879                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
880              }
881          }
882      );
883      menuAyuda13.add( elementoAyuda13 );
884
885      // crea el menú Herramientas y el elemento de menú Salir
886      JMenu menuHerramientas13 = new JMenu( "Herramientas" );
887      menuHerramientas13.setMnemonic( 'T' );
888      JMenuItem elementoHerramientas13 = new JMenuItem( "Herramientas" );
889      elementoHerramientas13.setMnemonic( 'h' );
890      elementoHerramientas13.addActionListener(
891          new ActionListener() {
892              public void actionPerformed( ActionEvent e )
893              {
894                  JOptionPane.showMessageDialog( PruebaMenu.this,
895                      "Este es un ejemplo\nde del uso de menus",
896                      "Herramientas", JOptionPane.PLAIN_MESSAGE );
897              }
898          }
899      );
900      menuHerramientas13.add( elementoHerramientas13 );
901
902      // crea el menú Edición y el elemento de menú Salir
903      JMenu menuEdicion13 = new JMenu( "Edición" );
904      menuEdicion13.setMnemonic( 'E' );
905      JMenuItem elementoEdicion13 = new JMenuItem( "Edición" );
906      elementoEdicion13.setMnemonic( 'e' );
907      elementoEdicion13.addActionListener(
908          new ActionListener() {
909              public void actionPerformed( ActionEvent e )
910              {
911                  JOptionPane.showMessageDialog( PruebaMenu.this,
912                      "Este es un ejemplo\nde del uso de menus",
913                      "Edición", JOptionPane.PLAIN_MESSAGE );
914              }
915          }
916      );
917      menuEdicion13.add( elementoEdicion13 );
918
919      // crea el menú Ver y el elemento de menú Salir
920      JMenu menuVer13 = new JMenu( "Ver" );
921      menuVer13.setMnemonic( 'V' );
922      JMenuItem elementoVer13 = new JMenuItem( "Ver" );
923      elementoVer13.setMnemonic( 'v' );
924      elementoVer13.addActionListener(
925          new ActionListener() {
926              public void actionPerformed( ActionEvent e )
927              {
928                  JOptionPane.showMessageDialog( PruebaMenu.this,
929                      "Este es un ejemplo\nde del uso de menus",
930                      "Ver", JOptionPane.PLAIN_MESSAGE );
931              }
932          }
933      );
934      menuVer13.add( elementoVer13 );
935
936      // crea el menú Ayuda y el elemento de menú Salir
937      JMenu menuAyuda14 = new JMenu( "Ayuda" );
938      menuAyuda14.setMnemonic( 'H' );
939      JMenuItem elementoAyuda14 = new JMenuItem( "Ayuda" );
940      elementoAyuda14.setMnemonic( 'y' );
941      elementoAyuda14.addActionListener(
942          new ActionListener() {
943              public void actionPerformed( ActionEvent e )
944              {
945                  JOptionPane.showMessageDialog( PruebaMenu.this,
946                      "Este es un ejemplo\nde del uso de menus",
947                      "Ayuda", JOptionPane.PLAIN_MESSAGE );
948              }
949          }
950      );
951      menuAyuda14.add( elementoAyuda14 );
952
953      // crea el menú Herramientas y el elemento de menú Salir
954      JMenu menuHerramientas14 = new JMenu( "Herramientas" );
955      menuHerramientas14.setMnemonic( 'T' );
956      JMenuItem elementoHerramientas14 = new JMenuItem( "Herramientas" );
957      elementoHerramientas1
```

```
47          } // fin del método actionPerformed
48      } // fin de la clase interna anónima
49  ); // fin de addActionListener
50 menuArchivo.add( elementoSalir );
51 barra.add( menuArchivo ); // agrega el menú Archivo
52
53 // crea el menú Formato, sus submenús y los elementos de menú
54 JMenu menuFormato = new JMenu( "Formato" );
55 menuFormato.setMnemonic( 'F' );
56
57 // crea el submenú Color
58 String colores[] =
59     { "Negro", "Azul", "Rojo", "Verde" };
60 JMenu menuColor = new JMenu( "Color" );
61 menuColor.setMnemonic( 'C' );
62 elementosColores = new JRadioButtonMenuItem[ colores.length ];
63 grupoColores = new ButtonGroup();
64 ManejadorElementos manejadorElementos = new ManejadorElementos();
65
66 for ( int i = 0; i < colores.length; i++ ) {
67     elementosColores[ i ] =
68         new JRadioButtonMenuItem( colores[ i ] );
69     menuColor.add( elementosColores[ i ] );
70     grupoColores.add( elementosColores[ i ] );
71     elementosColores[ i ].addActionListener( manejadorElementos );
72 } // fin de for
73
74 elementosColores[ 0 ].setSelected( true );
75 menuFormato.add( menuColor );
76 menuFormato.addSeparator();
77
78 // crea el submenú TipoDeLetra
79 String nombresTiposDeLetra[] =
80     { "TimesRoman", "Courier", "Helvetica" };
81 JMenu menuTipoDeLetra = new JMenu( "Fuente" );
82 menuTipoDeLetra.setMnemonic( 'T' );
83 tiposDeLetra = new JRadioButtonMenuItem[ nombresTiposDeLetra.length ];
84 grupoTiposDeLetra = new ButtonGroup();
85
86 for ( int i = 0; i < tiposDeLetra.length; i++ ) {
87     tiposDeLetra[ i ] =
88         new JRadioButtonMenuItem( nombresTiposDeLetra[ i ] );
89     menuTipoDeLetra.add( tiposDeLetra[ i ] );
90     grupoTiposDeLetra.add( tiposDeLetra[ i ] );
91     tiposDeLetra[ i ].addActionListener( manejadorElementos );
92 } // fin de for
93
94 tiposDeLetra[ 0 ].setSelected( true );
95 menuTipoDeLetra.addSeparator();
96
97 String nombresEstilos[] = { "Negrita", "Cursiva" };
98 elementosEstilo = new JCheckBoxMenuItem[ nombresEstilos.length ];
99 ManejadorEstilos manejadorEstilos = new ManejadorEstilos();
100
```

Figura 29.22 Uso de objetos **JMenu** y mnemónicos; **PruebaMenu.java**. (Parte 2 de 4.)

```
101     for ( int i = 0; i < nombresEstilos.length; i++ ) {
102         elementosEstilo[ i ] =
103             new JCheckBoxMenuItem( nombresEstilos[ i ] );
104         menuTipoDeLetra.add( elementosEstilo[ i ] );
105         elementosEstilo[ i ].addItemListener( manejadorEstilos );
106     } // fin de for
107
108     menuFormato.add( menuTipoDeLetra );
109     barra.add( menuFormato ); // agrega el menú Formato
110
111     pantalla = new JLabel(
112         "Texto muestra", SwingConstants.CENTER );
113     pantalla.setForeground( valoresColores[ 0 ] );
114     pantalla.setFont(
115         new Font( "TimesRoman", Font.PLAIN, 72 ) );
116
117     getContentPane().setBackground( Color.cyan );
118     getContentPane().add( pantalla, BorderLayout.CENTER );
119
120     setSize( 500, 200 );
121     show();
122 } // fin del constructor PruebaMenu
123
124 public static void main( String args[] )
125 {
126     PruebaMenu ap = new PruebaMenu();
127
128     ap.addWindowListener(
129         new WindowAdapter() {
130             public void windowClosing( WindowEvent e )
131             {
132                 System.exit( 0 );
133             } // fin del método windowClosing
134         } // fin de la clase interna anónima
135     ); // fin de addWindowListener
136 } // fin de main
137
138 class ManejadorElementos implements ActionListener {
139     public void actionPerformed( ActionEvent e )
140     {
141         for ( int i = 0; i < elementosColores.length; i++ )
142             if ( elementosColores[ i ].isSelected() ) {
143                 pantalla.setForeground( valoresColores[ i ] );
144                 break;
145             }
146
147         for ( int i = 0; i < tiposDeLetra.length; i++ )
148             if ( e.getSource() == tiposDeLetra[ i ] ) {
149                 pantalla.setFont( new Font(
150                     tiposDeLetra[ i ].getText(), estilo, 72 ) );
151                 break;
152             }
153
154         repaint();
155     }
156 }
```

Figura 29.22 Uso de objetos **JMenu** y mnemónicos; **PruebaMenu.java**. (Parte 3 de 4.)

```

155         } // fin del método actionPerformed
156     } // fin de la clase interna ManejadorElementos
157
158     class ManejadorEstilos implements ItemListener {
159         public void itemStateChanged( ItemEvent e ) {
160             {
161                 estilo = 0;
162
163                 if ( elementosEstilo[ 0 ].isSelected() )
164                     estilo += Font.BOLD;
165
166                 if ( elementosEstilo[ 1 ].isSelected() )
167                     estilo += Font.ITALIC;
168
169                 pantalla.setFont( new Font(
170                     pantalla.getFont().getName(), estilo, 72 ) );
171
172                 repaint();
173             } // fin del método itemStateChanged
174         } // fin de la clase interna ManejadorEstilos
175     } // fin de la clase PruebaMenu

```



Figura 29.22 Uso de objetos **JMenu** y mnemónicos; **PruebaMenu.java**. (Parte 4 de 4.)

La clase **PruebaMenu** (línea 7) es una clase completamente autocontenido: define todos los componentes GUI y el manejo de eventos para los elementos de menú. La mayor parte del código para esta aplicación aparece en el constructor de la clase (línea 16).

Las líneas 20 y 21

```

JMenuBar barra = new JMenuBar(); // crea la barra de menús
setJMenuBar( barra ); // establece la barra de menús para el objeto JFrame

```

crean el objeto **JMenuBar** y se adjunta a la ventana de la aplicación mediante el método **setJMenuBar** de **JFrame**.

Error común de programación 29.10



*Olvidar establecer la barra de menús con el método **setJMenuBar** de **JFrame** hará que la barra de menús no se despliegue en el objeto **JFrame**.*

Las líneas 24 a 51 establecen el menú **Archivo** y se adjunta a la barra de menús. Este menú contiene un elemento de menú llamado **Acerca de...**, el cual muestra un cuadro de diálogo de mensaje cuando se selecciona, y un elemento de menú llamado **Sair** que puede seleccionarse para terminar la aplicación.

La línea 24

```
JMenu menuArchivo = new JMenu( "Archivo" );
```

crea un objeto **JMenu**, se asigna a la referencia **menuArchivo**, y se pasa al constructor la cadena "**Archivo**" como el nombre del menú. La línea 25

```
menuArchivo.setMnemonic( 'A' );
```

utiliza el método **setMnemonic** de **AbstractButton** (heredado a la clase **JMenu**) para indicar que **A** es el *mnemónico* de este menú. Al oprimir la tecla *Alt* y la letra **A** se abre el menú, exactamente igual que al hacer clic en el nombre del menú con el ratón. En la GUI, el carácter mnemónico del nombre del menú aparece subrayado (vea las capturas de pantalla).

Observación de apariencia visual 29.25



Los mnemónicos proporcionan un acceso rápido con el teclado a los comandos de menú y de botón.

Observación de apariencia visual 29.26



Deben usarse distintos mnemónicos para cada botón o elemento de menú. En general, se utiliza la primera letra de la etiqueta correspondiente al elemento de menú o al botón como mnemónico. Si varios botones o elementos de menú empiezan con la misma letra, seleccione la siguiente letra más prominente en el nombre (por ejemplo, la letra **u** se utiliza comúnmente para un botón o elemento de menú llamado **Guardar como...**).

Las líneas 26 y 27

```
JMenuItem elementoAcercaDe = new JMenuItem( "Acerca de..." );
elementoAcercaDe.setMnemonic( 'c' );
```

definen el objeto **elementoAcercaDe** de **JMenuItem** con el nombre "**Acerca de...**" y establecen su mnemónico como la letra '**c**'. Este elemento de menú se agrega a **menuArchivo** en la línea 38. Para acceder al elemento **Acerca de...** por medio del teclado, oprima la tecla *Alt* y la letra **A** para abrir el menú **Archivo** y después oprima **c** para seleccionar el elemento de menú **Acerca de....**. Las líneas 28 a 37 crean un objeto **ActionListener** para escuchar la selección de **elementoAcercaDe**. Las líneas 32 a 34

```
JOptionPane.showMessageDialog( PruebaMenu.this,
    "Este es un ejemplo\nde uso de menus",
    "Acerca de", JOptionPane.PLAIN_MESSAGE );
```

muestran un cuadro de diálogo de mensaje. En la mayoría de las veces que utilizamos **showMessageDialog**, el primer argumento fue **null**. El propósito del primer argumento es especificar la *ventana padre* para el cuadro de diálogo. Esta ventana padre ayuda a determinar en dónde se va a desplegar el cuadro de diálogo. Si la ventana padre se especifica como **null**, el cuadro de diálogo se despliega en el centro de la pantalla. Si la ventana padre no es **null**, el cuadro de diálogo se despliega centrado horizontalmente sobre la ventana padre, y justo debajo de la parte superior de la ventana.

Los cuadros de diálogo pueden ser *modales* o *no modales*. Un *cuadro de diálogo modal* no permite el acceso a ninguna otra ventana de la aplicación, sino hasta que el cuadro de diálogo se cierra. Un *cuadro de diálogo no modal* permite el acceso a otras ventanas mientras éste se despliega en pantalla. De manera predeterminada, los cuadros de diálogo desplegados con la clase **JOptionPane** son cuadros de diálogo modales. Usted puede usar la clase **JDialog** para crear sus propios cuadros de diálogo modales o no modales.

La línea 38

```
menuArchivo.add( elementoAcercaDe );
```

agrega **elementoAcercaDe** al **menuArchivo** mediante el método **add** de **JMenu**.

Las líneas 40 a 50 definen el elemento de menú **elementoSair**, establecen su mnemónico como **S** y registran un objeto **ActionListener** que termina la aplicación cuando se selecciona **elementoSair**.

La línea 51

```
barra.add( menuArchivo ); // agrega el menú Archivo
```

utiliza el método **add** de **JMenuBar** para adjuntar el **menuArchivo** a **barra**.

Observación de apariencia visual 29.27



Los menús normalmente aparecen de izquierda a derecha, en el orden en el que se agregan.

Las líneas 54 y 55 crean el menú **menuFormato** y establecen su mnemónico como **F**.

Las líneas 60 y 61 crean el menú **menuColor** (éste será un submenú del menú **Formato**) y establecen su mnemónico como **C**. La línea 62 crea el arreglo **JRadioButtonMenuItem** llamado **elementosColores**, el cual hará referencia a los elementos de menú que se encuentran en **menuColor**. La línea 63 crea el objeto **ButtonGroup** llamado **grupoColores**, el cual se asegurará de que solamente se seleccione uno de los elementos de menú del submenú **Color** en un momento dado. La línea 64 define una instancia de la clase interna **ManejadorElementos** (definida en las líneas 138 a 156), la cual se utilizará para responder a las selecciones de los submenús **Color** y **Fuente** (que describiremos en breve). La estructura **for** de las líneas 66 a 72 crea cada objeto **JRadioButtonMenuItem** en el arreglo **elementosColores**, agrega cada elemento de menú a **menuColor**, agrega cada elemento de menú a **grupoColores** y registra el objeto **ActionListener** para cada elemento de menú.

La línea 74

```
elementosColores[ 0 ].setSelected( true );
```

utiliza el método **setSelected** de **AbstractButton** para indicar que el primer elemento del arreglo **elementosColores** debe estar seleccionado. La línea 75 agrega el **menuColor** como un submenú del **menuFormato**.

Observación de apariencia visual 29.28



Agregar un menú como elemento de otro menú lo convierte automáticamente en un submenú. Cuando el ratón se coloca sobre un submenú (o cuando se oprime el mnemónico de ese submenú), éste se expande para mostrar sus elementos.

La línea 76

```
menuFormato.addSeparator();
```

agrega una línea *separadora* al menú. El separador aparece como una línea horizontal en el menú.

Observación de apariencia visual 29.29



Es posible agregar separadores a un menú para agrupar los elementos en forma lógica.

Las líneas 79 a 94 crean el submenú **Fuente** y varios objetos **JRadioButtonMenuItem**, e indican que el primer elemento del arreglo de objetos **JRadioButtonMenuItem** llamado **tiposDeLetra** debe estar seleccionado. La línea 98 crea un arreglo de objetos **JCheckBoxMenuItem** para representar los elementos de menú para especificar los estilos negrita y cursiva para la fuente. La línea 99 define una instancia de la clase interna **ManejadorEstilos** (definida en las líneas 158 a 174) para responder a los eventos de **JCheckBoxMenuItem**. La estructura **for** de las líneas 101 a 106 crea cada objeto **JCheckBoxMenuItem**, agrega cada elemento de menú a **menuTipoDeLetra** y registra el objeto **ItemListener** para cada elemento de menú. La línea 108 agrega **menuTipoDeLetra** como un submenú de **menuFormato**. La línea 109 agrega el **menuFormato** a **barra**.

Las líneas 111 a 115 crean un objeto **JLabel** en el que la fuente, el color y el estilo se controlan a través del menú **Formato**. El color inicial de primer plano se establece como el primer elemento del arreglo **valoresColores** (**Color.black**) y el tipo de letra inicial se establece como **TimesRoman** con estilo **PLAIN** y tamaño de 72 puntos. La línea 117 establece el color de fondo del panel de contenido de la ventana como **Color.cyan**, y la línea 118 adjunta el objeto **JLabel** a la región **CENTER** del diseño **BorderLayout** del panel de contenido.

El método `actionPerformed` de la clase `ManejadorElementos` (línea 138) utiliza dos estructuras `for` para determinar cuál elemento de menú fuente o color generó el evento, y establece la fuente o el color del objeto `pantalla` de `JLabel`, respectivamente. La condición `if` de la línea 142 utiliza el método `isSelected` de `AbstractButton` para determinar cuál objeto `JRadioButtonMenuItem` para seleccionar colores está seleccionado. La condición `if` de la línea 148 utiliza el método `getSource` de `EventSource` para obtener una referencia al objeto `JRadioButtonMenuItem` que generó el evento. La línea 150 utiliza el método `getText` de `AbstractButton` para obtener el nombre del tipo de letra, del elemento de menú.

El método `itemStateChanged` de la clase `ManejadorEstilos` (línea 158) se llama si el usuario selecciona un objeto `JCheckBoxMenuItem` en el `menuTipoDeLetra`. Las líneas 163 y 166 determinan si uno o ambos objetos `JCheckBoxMenuItem` están seleccionados, y utiliza su estado combinado para determinar el nuevo estilo de la fuente.

Observación de apariencia visual 29.30



Cualquier componente GUI ligero (es decir, un componente que sea subclase de `JComponent`) puede agregarse a un objeto `JMenu` o `JMenuBar`.

RESUMEN

- Una interfaz gráfica de usuario (GUI) presenta una interfaz ilustrada de un programa. Una GUI proporciona a un programa una “apariencia visual” única.
- Al proporcionar a distintas aplicaciones un conjunto consistente de componentes intuitivos de la interfaz del usuario, las GUIs permiten al usuario pasar más tiempo utilizando el programa de una manera más productiva.
- Las GUIs se crean a partir de componentes GUI (algunas veces conocidos como controles o “widgets”). Un componente GUI es un objeto visual con el que el usuario interactúa mediante el ratón o el teclado.
- Los componentes GUI de Swing están definidos en el paquete `javax.swing`. Los componentes Swing están escritos, se manipulan y se despliegan completamente en Java.
- Los componentes GUI originales del paquete `java.awt` del Abstract Windowing Toolkit están enlazados directamente con las herramientas de la interfaz gráfica de usuario de la plataforma local.
- Los componentes de Swing son componentes ligeros. Los componentes del AWT están enlazados a la plataforma local y algunas veces se les conoce como componentes pesados: dependen del sistema de ventanas de la plataforma local para determinar su funcionalidad y su apariencia visual.
- Varios componentes GUI de Swing son componentes GUI pesados: en especial, las subclases de `java.awt.Window` (como `JFrame`) que muestran ventanas en la pantalla. Los componentes GUI pesados de Swing son menos flexibles que los componentes ligeros.
- La mayor parte de las herramientas de cada componente GUI de Swing se hereda de las clases `Component`, `Container` y `JComponent` (la superclase para la mayoría de los componentes Swing).
- Un objeto `Container` es un área en la que pueden colocarse componentes.
- Los objetos `JLabel` proporcionan instrucciones o información textual en una GUI.
- El método `setToolTipText` de `JComponent` especifica la información de herramienta que se despliega siempre que el usuario posiciona el cursor del ratón sobre un objeto `JComponent` en la GUI.
- Muchos componentes Swing pueden mostrar imágenes especificando un objeto `Icon` como argumento para su constructor, o utilizando un método `setIcon`.
- La clase `ImageIcon` (del paquete `javax.swing`) soporta dos formatos de imagen: el Formato de intercambio de gráficos (GIF) y el Grupo unido de expertos en fotografía (JPEG).
- La interfaz `SwingConstants` (del paquete `javax.swing`) define un conjunto de constantes enteras comunes (como `SwingConstants.LEFT`) que se utilizan con muchos componentes Swing.
- De manera predeterminada, el texto de un objeto `JComponent` aparece a la derecha de la imagen, cuando el objeto `JComponent` contiene tanto texto como una imagen.
- Las alineaciones horizontal y vertical de un objeto `JLabel` pueden establecerse mediante los métodos `setHorizontalAlignment` y `setVerticalAlignment`. El método `setText` establece el texto que se despliega en la etiqueta. El método `getText` recupera el texto actual que aparece en una etiqueta. Los métodos `setHorizontalTextPosition` y `setVerticalTextPosition` especifican la posición del texto en una etiqueta.

- El método **setIcon** de **JComponent** establece el objeto **Icon** que va a mostrarse en un objeto **JComponent**. El método **getIcon** recupera el objeto **Icon** actual mostrado en un objeto **JComponent**.
- Las GUIs generan eventos cuando el usuario interactúa con ellas. La información acerca de un evento GUI se almacena en un objeto de una clase que extienda a **AWTEvent**.
- Para procesar un evento, el programador debe registrar un componente que escuche eventos, e implementar uno o más manejadores de eventos.
- Al uso de componentes de escucha en el manejo de eventos se le conoce como modelo de delegación de eventos: el procesamiento de un evento se delega a un objeto específico en el programa.
- Cuando ocurre un evento, el componente GUI con el que interactuó el usuario notifica a sus componentes de escucha registrados mediante una llamada al método manejador de eventos apropiado para cada componente de escucha.
- Los objetos **JTextField** y **JPasswordField** son áreas de una sola línea en las que el usuario puede introducir texto desde el teclado, o simplemente pueden mostrar texto. Un objeto **JPasswordField** muestra que se escribió un carácter a medida que el usuario va escribiendo, pero oculta automáticamente los caracteres.
- Cuando el usuario escribe datos en un objeto **JTextField** o **JPasswordField** y oprime *Entrar*, se genera un evento **ActionEvent**.
- El método **setEditable** de **JTextComponent** determina si el usuario puede modificar el texto de un objeto **JTextComponent**.
- El método **getPassword** de **JPasswordField** devuelve la contraseña como un arreglo de tipo **char**.
- Cada objeto **JComponent** contiene un objeto de la clase **EventListenerList** (del paquete **javax.swing.event**) llamado **listenerList**, en el que se almacenan todos los componentes de escucha registrados.
- Cada objeto **JComponent** soporta varios tipos de eventos distintos, que incluyen eventos de ratón, de tecla y otros más. Cuando ocurre un evento, éste se despacha (se envía) solamente a los componentes de escucha de eventos del tipo apropiado. Cada tipo de evento tiene su interfaz de escucha de eventos correspondiente.
- Cuando se genera un evento debido a la interacción de un usuario con un componente, al componente se le otorga un ID de evento único para especificar el tipo de evento. El componente GUI utiliza el ID de evento para decidir el tipo de componente de escucha al que debe despacharse el evento, junto con el método manejador de eventos al que debe llamar.
- Un objeto **JButton** genera un evento **ActionEvent** cuando el usuario hace clic en el botón con el ratón.
- Un objeto **AbstractButton** puede tener un objeto **Icon** de sustitución que se despliega cuando el ratón se coloca sobre el botón. El icono cambia a medida que el ratón se desplaza hacia adentro y hacia afuera del área del botón en la pantalla. El método **setRollOverIcon** de **AbstractButton** especifica la imagen a desplegar en un botón, cuando el usuario coloca el ratón sobre ese botón.
- Los componentes GUI de Swing contienen tres tipos de botones de estado (**JToggleButton**, **JCheckBox** y **JRadioButton**) con valores de encendido/apagado o verdadero/falso. Las clases **JCheckBox** y **JRadioButton** son subclases de **JToggleButton**.
- Cuando el usuario hace clic en un objeto **JCheckBox** se genera un evento **ItemEvent**, el cual puede ser manejado por un objeto **ItemListener**. Estos objetos deben definir al método **itemStateChanged**. El método **getStateChange** de **ItemEvent** determina el estado de un objeto **JToggleButton**.
- Los objetos **JRadioButton** son similares a los objetos **JCheckBox** en cuanto a que tienen dos estados: seleccionado y no seleccionado. Los objetos **JRadioButton** generalmente aparecen como un grupo en el que sólo puede haber un botón de opción seleccionado a la vez.
- Un objeto **JComboBox** (al que algunas veces se le conoce como lista desplegable) proporciona una lista de elementos para que el usuario seleccione uno de ellos. Los objetos **JComboBox** generan eventos **ItemEvent**. Un índice numérico lleva el registro del orden de los elementos en un objeto **JComboBox**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento agregado a un objeto **JComboBox** aparece como el elemento actualmente seleccionado cuando se despliega el objeto **JComboBox**. El método **getSelectedIndex** de **JComboBox** devuelve el número del índice correspondiente al elemento seleccionado.
- Es posible atrapar eventos de ratón para cualquier componente GUI que se derive de **java.awt.Component** por medio de objetos **MouseListener** y **MouseMotionListener**.
- Cada método manejador de eventos de ratón toma como su argumento un objeto **MouseEvent** que contiene información acerca del evento de ratón y la ubicación en donde ocurrió el evento.
- Los métodos **addMouseListener** y **addMouseMotionListener** son métodos de **Component** utilizados para registrar componentes que escuchan eventos de ratón para un objeto de cualquier clase que extienda a **Component**.

- Muchas de las interfaces que escuchan eventos proporcionan varios métodos. Para cada uno de ellos hay su correspondiente clase adaptadora de escucha de eventos, la cual proporciona una implementación detallada de cada método en la interfaz. El programador puede extender la clase adaptadora para heredar la implementación predeterminada de cada método y simplemente redefinir el método o métodos necesarios para el manejo de eventos en el programa.
- El método `getClickCount` de `MouseEvent` devuelve el número de clics del ratón.
- Los métodos `isMetaDown` e `isAltDown` de `InputEvent` se utilizan para determinar en qué botón hizo clic el usuario.
- Los administradores de diseños ordenan los componentes GUI en un contenedor para fines de presentación.
- `FlowLayout` distribuye los componentes de izquierda a derecha, en el orden en el que se agregan al contenedor. Al llegar al borde del contenedor, los componentes continúan en la siguiente línea.
- El método `setAlignment` de `FlowLayout` cambia la alineación del diseño `FlowLayout` a `FlowLayout.LEFT`, `FlowLayout.CENTER` o `FlowLayout.RIGHT`.
- El administrador de diseño `BorderLayout` ordena los componentes en cinco regiones: Norte, Sur, Este, Oeste y Centro. Puede agregarse un componente a cada región.
- El método `layoutContainer` de `LayoutManager` recalculara la distribución de su argumento `Container`.
- El administrador de diseño `GridLayout` divide el contenedor en una cuadrícula de filas y columnas. Los componentes se agregan a un diseño `GridLayout`, empezando en la celda superior izquierda y procediendo de izquierda a derecha, hasta que la fila esté llena. Después, el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, etcétera.
- El método `validate` de `Container` recalculara la distribución del contenedor con base en el administrador de diseño actual para el objeto `Container` y el conjunto actual de componentes GUI desplegados en pantalla.
- Los paneles se crean mediante la clase `JPanel`, la cual hereda de la clase `JComponent`. Se pueden agregar componentes a los objetos `JPanel`, incluso otros paneles.
- Los objetos `JTextArea` proporcionan un área para manipular varias líneas de texto. Al igual que la clase `JTextField`, la clase `JTextArea` hereda de `JTextComponent`.
- Un evento externo (es decir, un evento generado por un componente GUI distinto) generalmente indica cuándo debe procesarse el texto en un objeto `JTextArea`.
- Para un objeto `JTextArea` se proporcionan barras de desplazamiento, si éste se adjunta a un objeto `JScrollPane`.
- El método `getSelectedText` devuelve el texto seleccionado de un objeto `JTextArea`. El texto se selecciona arrastrando el ratón sobre el texto deseado para resaltarlo.
- El método `setText` establece el texto en un objeto `JTextArea`.
- Para proporcionar la envoltura automática de palabras en un objeto `JTextArea`, adjúntelo a un objeto `JScrollPane` con la directiva de barra de desplazamiento horizontal `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`.
- Las directivas de barras de desplazamiento horizontal y vertical para un objeto `JScrollPane` se establecen cuando se crea, o por medio de los métodos `setHorizontalScrollBarPolicy` y `setVerticalScrollBarPolicy` de la clase `JScrollPane`.
- Un objeto `JPanel` puede utilizarse como área dedicada de dibujo, la cual puede recibir eventos de ratón y a menudo se extiende para crear nuevos componentes GUI.
- Los componentes Swing que heredan de la clase `JComponent` contienen el método `paintComponent`, el cual los ayuda a dibujar adecuadamente dentro del contexto de una GUI de Swing. El método `paintComponent` de `JComponent` debe redefinirse de la siguiente manera:

```
public void paintComponent( Graphics g )
{
    super.paintComponent( g );
    // el código adicional de dibujo
}
```

- La llamada a la versión de `paintComponent` correspondiente a la superclase garantiza que la acción de dibujar ocurra en el orden adecuado y que el mecanismo de dibujo de Swing permanezca intacto. Si no se hace una llamada a la versión de `paintComponent` correspondiente a la superclase, por lo general el componente GUI personalizado no se desplegará apropiadamente en la interfaz de usuario. Además, si se hace la llamada a la versión de la superclase después de ejecutar las instrucciones de dibujo personalizadas, los resultados por lo general se borran.

- Las clases **JFrame** y **JApplet** no son subclases de **JComponent**; por lo tanto, no contienen el método **paintComponent** (tienen el método **paint**).
- Al llamar a **repaint** para un componente GUI de Swing se indica que el componente deberá dibujarse lo más pronto posible. El fondo del componente GUI se borra solamente si el componente es opaco. De manera predeterminada, la mayoría de los componentes Swing son transparentes. El método **setOpaque** de **JComponent** puede recibir un argumento **booleano** que indica si el componente es opaco (**true**) o transparente (**false**). Los componentes GUI del paquete **java.awt** son distintos de los componentes Swing, en cuanto a que **repaint** produce una llamada al método **update** de **Component** (el cual borra el fondo del componente) y **update** a su vez llama al método **paint** (en lugar de llamar a **paintComponent**).
- El método **setTitle** despliega un objeto **String** en la barra de título de una ventana.
- Para dibujar en cualquier componente GUI se requieren coordenadas que se miden a partir de la esquina superior izquierda (**0, 0**) de ese componente GUI.
- Los administradores de diseño a menudo utilizan el método **getPreferredSize** de un componente GUI para determinar los mejores valores para el ancho y la altura, al distribuir ese componente como parte de una GUI. Si un nuevo componente tiene un mejor valor de ancho y altura, debe redefinirse el método **getPreferredSize** para que devuelva ese ancho y esa altura como un objeto de la clase **Dimension** (del paquete **java.awt**).
- El tamaño predeterminado de un objeto **JPanel** es de 0 pixeles de ancho y 0 pixeles de alto.
- Una operación de arrastre de ratón empieza con un evento de botón oprimido del ratón. Todos los eventos subsecuentes de arrastre del ratón (para los cuales se hará una llamada a **mouseDragged**) se envían al componente GUI que recibió el evento original de botón oprimido del ratón.
- Un objeto **JFrame** es una ventana con una barra de título y un borde. La clase **JFrame** es una subclase de **java.awt.Frame** (que a su vez es una subclase de **java.awt.Window**).
- La clase **JFrame** soporta tres operaciones cuando el usuario cierra la ventana. De manera predeterminada, cuando el usuario cierra una ventana, el objeto **JFrame** se oculta. Esto puede controlarse mediante el método **setDefaultCloseOperation** de **JFrame**. La interfaz **WindowConstants** (del paquete **javax.swing**) define tres constantes para usarse con este método: **DISPOSE_ON_CLOSE**, **DO NOTHING ON CLOSE** y **HIDE ON CLOSE** (la opción predeterminada).
- De manera predeterminada, una ventana no se despliega en la pantalla sino hasta que se llama a su método **show**. Una ventana también puede desplegarse llamando a su método **setVisible**, con **true** como argumento.
- El tamaño de una ventana debe establecerse mediante una llamada al método **setSize**. La posición que tendrá una ventana al aparecer en pantalla se especifica mediante el método **setLocation**.
- Todas las ventanas generan eventos de ventana cuando el usuario las manipula. Los componentes que escuchan eventos se registran para los eventos de ventana por medio del método **addWindowListener** de la clase **Window**. La interfaz **WindowListener** proporciona siete métodos para manejar eventos de ventana: **windowActivated** (se llama cuando la ventana se convierte en la ventana activa al hacer clic en ella), **windowClosed** (se llama después de cerrar la ventana), **windowClosing** (se llama cuando el usuario inicia el proceso de cerrar la ventana), **windowDeactivated** (se llama cuando otra ventana se convierte en la ventana activa), **windowIconified** (se llama cuando el usuario minimiza una ventana), **windowDeiconified** (se llama cuando una ventana se restaura, después de estar minimizada) y **windowOpened** (se llama cuando se muestra una ventana por primera vez en la pantalla).
- Los argumentos de línea de comandos se pasan automáticamente a **main** como el arreglo de objetos **String** llamado **args**. El primer argumento después del nombre de la clase de la aplicación es el primer objeto **String** del arreglo **args**, y la longitud del arreglo es el número total de argumentos de la línea de comandos.
- Los menús son una parte integral de las GUIs, ya que permiten al usuario realizar acciones sin “atestar” innecesariamente una interfaz gráfica de usuario con componentes GUI adicionales.
- En las GUIs de Swing, los menús sólo pueden adjuntarse a objetos de las clases que proporcionan el método **setJMenuBar**. Dos de esas clases son **JFrame** y **JApplet**.
- Las clases utilizadas para definir menús son **JMenuBar**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem** y **JRadioButtonMenuItem**.
- Un objeto **JMenuBar** es un contenedor de menús.
- Un objeto **JMenuItem** es un componente GUI dentro de un menú que, cuando se selecciona, hace que se lleve a cabo cierta acción. Un objeto **JMenuItem** puede usarse para iniciar una acción o puede ser un submenú que proporcione más elementos de menú que el usuario pueda seleccionar.

- Un objeto **JMenu** contiene elementos de menú y puede agregarse a un objeto **JMenuBar** o a otros objetos **JMenu** como submenú. Al hacer clic en un menú, éste se expande para mostrar su lista de elementos.
- Al seleccionar un objeto **JCheckBoxMenuItem** aparece una marca de verificación a la izquierda del elemento de menú. Cuando se selecciona nuevamente este objeto **JCheckBoxMenuItem**, la marca de verificación desaparece.
- Cuando se mantienen varios objetos **JRadioButtonMenuItem** como parte de un objeto **ButtonGroup**, sólo puede seleccionarse un elemento del grupo a la vez. Cuando se selecciona un objeto **JRadioButtonMenuItem**, aparece un círculo relleno a la izquierda del elemento de menú. Cuando se selecciona otro **JRadioButtonMenuItem**, se quita el círculo relleno a la izquierda del elemento de menú previamente seleccionado.
- El método **setJMenuBar** de **JFrame** adjunta una barra de menús a un objeto **JFrame**.
- El método **setMnemonic** de **AbstractButton** (heredado en la clase **JMenu**) especifica el mnemónico para un objeto **AbstractButton**. Al oprimir la tecla *Alt* y el mnemónico se lleva a cabo la acción del objeto **AbstractButton** (en el caso de un menú, éste se abre).
- Los caracteres mnemónicos normalmente aparecen subrayados.
- Los cuadros de diálogo pueden ser modales o no modales. Un cuadro de diálogo modal no permite el acceso a ninguna otra ventana en la aplicación, sino hasta que el cuadro de diálogo se cierra. Un cuadro de diálogo no modal permite el acceso a otras ventanas mientras se despliega en pantalla. De manera predeterminada, los cuadros de diálogo que se despliegan mediante la clase **JOptionPane** son cuadros de diálogo modales. La clase **JDialog** puede usarse para crear cuadros de diálogo modales o no modales.
- El método **addSeparator** de **JMenu** agrega una línea separadora a un menú.

TERMINOLOGÍA

Abstract Windows Toolkit	clase GridLayout	cuadro de diálogo no modal
administrador de diseño	clase ImageIcon	despachar un evento
administrador de diseño	clase ItemEvent	directivas de barra de desplazamiento para un objeto
BoxLayout	clase JButton	JScrollPane
alineado a la derecha	clase JCheckBox	elemento de menú
alineado a la izquierda	clase JCheckBoxMenuItem	envoltura automática de palabras
área de dibujo dedicada	clase JComboBox	“escuchar” un evento
arrastrar	clase JComponent	espacio libre horizontal
barra de desplazamiento	clase JLabel	espacio libre vertical
barra de herramientas	clase JMenu	etiqueta
barra de menús	clase JMenuBar	etiqueta de botón
BorderLayout.CENTER	clase JMenuItem	etiqueta de casilla de verificación
BorderLayout.EAST	clase JPanel	evento
BorderLayout.NORTH	clase JPasswordField	evento externo
BorderLayout.SOUTH	clase JRadioButton	extensión .gif de nombre de archivo
BorderLayout.WEST	clase JScrollPane	extensión .jpg de nombre de archivo
botón	clase JTextArea	flecha de desplazamiento
botón de comando	clase JTextComponent	FlowLayout.CENTER
botón de opción	clase JTextField	FlowLayout.LEFT
casilla de verificación	clase JToggleButton	FlowLayout.RIGHT
clase AbstractButton	clase MouseAdapter	Font.BOLD
clase ActionEvent	clase MouseEvent	Font.ITALIC
clase adaptadora	clase MouseMotionAdapter	Font.PLAIN
clase BorderLayout	clase WindowAdapter	Formato de intercambio de gráficos (GIF)
clase Box	componente GUI	Grupo unido de expertos en fotografía (JPEG)
clase Component	componente GUI de Swing	ícono de sustitución
clase ComponentAdapter	componente ligero	ID de evento
clase Container	componente pesado	información de herramientas
clase ContainerAdapter	componente que escucha eventos	
clase EventListenerList	control	
clase EventObject	controlado por eventos	
clase FlowLayout	cuadro de desplazamiento	
clase FocusAdapter	cuadro de diálogo modal	

interfaz ActionListener	método getSelectedIndex de JList	método setSelectionMode
interfaz ComponentListener	método getSelectedText	método setText
interfaz ContainerListener	método getSelectedValues de JList	método setTitle de la clase Frame
interfaz FocusListener	método getSource de ActionEvent	método setToolTipText
interfaz Icon	método getStateChange de ItemEvent	método setVertical- Alignment
interfaz ItemListener	método getText de JLabel	método setVerticalScroll- BarPolicy
interfaz LayoutManager	método getX de MouseEvent	método setVerticalText- Position
interfaz MouseListener	método getY de MouseEvent	método setVisible
interfaz MouseMotionListener	método itemStateChanged	método setVisibleRowCount
interfaz que escucha eventos	método layoutContainer	método validate
interfaz SwingConstants	método mouseClicked	método valueChanged
interfaz windowListener	método mouseDragged	método windowActivated
ItemEvent.DESELECTED	método mouseEntered	método windowClosed
ItemEvent.SELECTED	método mouseExited	método windowClosing
justificado a la izquierda	método mouseMoved	método windowDeactivated
lista de selección múltiple	método mousePressed	método windowDeiconified
lista de selección simple	método mouseReleased	método windowOpened
lista desplegable	método paintComponent	mnemónico
localización de la interfaz de usuario	de JComponent	modelo de delegación de eventos
manejador de eventos	método setAlignment	modo de selección
menú	método setBackground	paquete java.awt
método actionPerformed	método setDefaultClose- Operation	paquete java.awt.event
método add de la clase Container	método setEditable	paquete javax.swing
método addItemListener	método setHorizontal- ScrollBarPolicy	paquete javax.swing.event
método addMouseListener	método setIcon	registrar un componente que
método addMouseMotion- Listener	método setJMenuBar	escucha eventos
método addSeparator de la	método setLayout de la clase Container	sistema de ventanas
clase JMenu	método setListData de JList	submenú
método addWindowListener	método setMaximumRowCount	SwingConstants.HORIZONTAL
de Window	método setMnemonic de AbstractButton	SwingConstants.VERTICAL
método dispose de la clase	método setOpaque de la clase JComponent	tecla de método abreviado (mnemónicos)
Window	método setRollOverIcon	texto de sólo lectura
método getActionCommand	método setSelected de	ventana
método getIcon	AbstractButton	widgets o controles (accesorios de ventana)
método getPreferredSize de Component		windowConstants. DISPOSE_ON_CLOSE
método getPassword de JPasswordField		
método getPreferredSize de Component		
método getSelectedIndex de JComboBox		

ERRORES COMUNES DE PROGRAMACIÓN

- 29.1 Olvidar agregar un componente a un contenedor, para que pueda mostrarse en pantalla, es un error lógico en tiempo de ejecución.
- 29.2 Si se agrega a un contenedor un componente que no se haya instanciado, se lanza una excepción **NullPointerException**.
- 29.3 Utilizar una letra **f** minúscula en los nombres de las clases **JTextField** o **JPasswordField**, es un error de sintaxis.
- 29.4 Olvidar registrar un objeto manejador de eventos para un tipo de evento de un componente GUI en particular, da como resultado que no se manejen los eventos de ese componente.
- 29.5 Si se agrega más de un componente a una región específica en un diseño **BorderLayout**, sólo se desplegará el último componente que se haya agregado. No hay un mensaje de error para indicar este problema.

- 29.6 Cuando se redefine el método **paintComponent** de un objeto **JComponent**, si no se hace una llamada a la versión original de **paintComponent** de la superclase, el componente GUI no podrá desplegarse apropiadamente en la GUI.
- 29.7 Cuando se redefine el método **paintComponent** de un objeto **JComponent**, al llamar a la versión original de **paintComponent** de la superclase después de realizar otro dibujo, se borran los demás dibujos.
- 29.8 Olvidar llamar al método **show** o al método **setVisible** en una ventana, es un error lógico en tiempo de ejecución; la ventana no se desplegará en pantalla.
- 29.9 Olvidar llamar al método **setSize** en una ventana, es un error lógico en tiempo de ejecución; sólo aparecerá la barra de título.
- 29.10 Olvidar establecer la barra de menús con el método **setJMenuBar** de **JFrame** hará que la barra de menús no se despliegue en el objeto **JFrame**.

BUENAS PRÁCTICAS DE PROGRAMACIÓN

- 29.1 Estudie los métodos de la clase **Component** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de la mayoría de los componentes GUI.
- 29.2 Estudie los métodos de la clase **Container** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.
- 29.3 Estudie los métodos de la clase **JComponent** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas comunes de todos los contenedores de componentes GUI.
- 29.4 Estudie los métodos de la clase **javax.swing.JLabel** que se encuentran en la documentación en línea del SDK de Java 2, para que aprenda acerca de las herramientas completas de la clase antes de usarla.
- 29.5 Utilice clases separadas para procesar eventos GUI.

OBSERVACIONES DE APARIENCIA VISUAL

- 29.1 Las interfaces de usuario consistentes permiten a un usuario aprender a utilizar nuevas aplicaciones en menos tiempo.
- 29.2 Los componentes Swing están escritos en Java, por lo que ofrecen un mayor nivel de portabilidad y flexibilidad que los componentes GUI originales de Java del paquete **java.awt**.
- 29.3 Utilice los cuadros de información de herramienta (establecidos mediante el método **setToolTipText** de **JComponent**) para agregar texto descriptivo a sus componentes GUI. Este texto ayuda al usuario a determinar el propósito del componente GUI en la interfaz de usuario.
- 29.4 A menudo, un evento externo determina cuándo debe procesarse el texto de un objeto **JTextArea**.
- 29.5 Para proporcionar la funcionalidad de envoltura automática de palabras para un objeto **JTextArea**, invoque al método **setLineWrap** con un argumento **true**.
- 29.6 Tener más de un objeto **JButton** con la misma etiqueta hace que los objetos **JButton** sean ambiguos para el usuario. Asegúrese de proporcionar una etiqueta única para cada botón.
- 29.7 El uso de iconos de sustitución para objetos **JButton** proporciona al usuario una retroalimentación visual, la cual le indica que, si hace clic en el ratón, se realizará la acción del botón.
- 29.8 La clase **AbstractButton** soporta que se despliegue texto e imágenes en un botón, por lo que todas las subclases de **AbstractButton** también soportan el despliegue de texto e imágenes.
- 29.9 Establezca el conteo máximo de filas para un objeto **JComboBox** en un número que evite que la lista se expanda más allá de los límites de la ventana o del applet en que se utilice. Esto garantizará que la lista aparezca correctamente cuando el usuario la expanda.
- 29.10 Las llamadas al método **mouseDragged** se envían al objeto **MouseMotionListener** para el objeto **Component** en el que se inició la operación de arrastre. De manera similar, la llamada al método **mouseReleased** se envía al objeto **MouseListener** para el objeto **Component** en el que se inició la operación de arrastre.
- 29.11 La mayoría de los entornos de programación de Java proporcionan herramientas de diseño GUI, las cuales ayudan a un programador a diseñar de manera gráfica una GUI, y después escriben automáticamente el código de Java necesario para crear la GUI.
- 29.12 Cada contenedor puede tener solamente un administrador de diseño a la vez (varios contenedores en el mismo programa pueden tener distintos administradores de diseño).

- 29.13** Si no se especifica una región al agregar un objeto **Component** a un diseño **BorderLayout**, se asume que el objeto **Component** va a agregarse a la región **BorderLayout.CENTER**.
- 29.14** Combinar gráficos y componentes GUI puede ocasionar un despliegue incorrecto de los gráficos, de los componentes GUI o de ambos. Utilizar objetos **JPanel** para dibujar puede eliminar este problema, proporcionando un área de dibujo dedicada para los gráficos.
- 29.15** Cuando se redefine el método **paintComponent** de un objeto **JComponent**, la primera instrucción del cuerpo siempre debe ser una llamada a la versión original del método de la superclase.
- 29.16** Llamar a **repaint** para un componente GUI de Swing indica que ese componente debe pintarse lo más pronto posible. El fondo del componente GUI se borra solamente si el componente es opaco. La mayoría de los componentes Swing son transparentes de manera predeterminada. Es posible pasar un argumento booleano al método **setOpaque** de **JComponent** para indicar si el componente es opaco (**true**), o transparente (**false**). Los componentes GUI del paquete **java.awt** son distintos de los componentes Swing en cuanto a que **repaint** produce una llamada al método **update** de **Component** (con lo cual se borra el fondo del componente), y **update**, a su vez, llama al método **paint** (en lugar de llamar a **paintComponent**).
- 29.17** El proceso de dibujar en cualquier componente GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0, 0) de ese componente GUI.
- 29.18** El tamaño predeterminado de un objeto **JPanel** es de 0 pixeles de ancho y de 0 pixeles de alto.
- 29.19** Al crear subclases de **JPanel** (o de cualquier otro **JComponent**), se debe redefinir el método **getPreferredSize** si el nuevo componente debe tener mejores valores para el ancho y la altura.
- 29.20** La mayoría de los componentes Swing pueden ser transparentes u opacos. Si un componente GUI de Swing es opaco, al llamar a su método **paintComponent** su fondo se borrará; en caso contrario, no se borrará.
- 29.21** La clase **JComponent** proporciona el método **setOpaque** que toma un argumento booleano para determinar si un objeto **JComponent** es opaco (**true**) o transparente (**false**).
- 29.22** Los objetos **JPanel** son opacos de manera predeterminada.
- 29.23** Una operación de arrastre de ratón empieza con un evento de oprimir el botón del ratón (**mousePressed**). Todos los eventos subsecuentes de arrastre del ratón (para los cuales se hará una llamada a **mouseDragged**) se envían al componente GUI que recibió el evento original del botón oprimido del ratón.
- 29.24** Los menús simplifican las GUIs, al reducir el número de componentes que ve el usuario.
- 29.25** Los mnemónicos proporcionan un acceso rápido con el teclado a los comandos de menú y de botón.
- 29.26** Deben usarse distintos mnemónicos para cada botón o elemento de menú. En general, se utiliza la primera letra de la etiqueta correspondiente al elemento de menú o al botón como mnemónico. Si varios botones o elementos de menú empiezan con la misma letra, seleccione la siguiente letra más prominente en el nombre (por ejemplo, la letra **U** se utiliza comúnmente para un botón o elemento de menú llamado **Guardar como...**).
- 29.27** Los menús normalmente aparecen de izquierda a derecha, en el orden en el que se agregan.
- 29.28** Agregar un menú como elemento de otro menú lo convierte automáticamente en un submenú. Cuando el ratón se coloca sobre un submenú (o cuando se oprime el mnemónico de ese submenú), éste se expande para mostrar sus elementos.
- 29.29** Es posible agregar separadores a un menú para agrupar los elementos en forma lógica.
- 29.30** Cualquier componente GUI ligero (es decir, un componente que sea subclase de **JComponent**) puede agregarse a un objeto **JMenu** o **JMenuBar**.

TIP DE PORTABILIDAD

- 29.1** La apariencia de una GUI definida con componentes GUI pesados del paquete **java.awt** puede variar entre plataformas. Los componentes pesados se “enlanzan” a la GUI de la plataforma “local”, la cual varía entre las distintas plataformas.

OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 29.1** Para utilizar componentes GUI con efectividad debe comprender las jerarquías de herencia de **javax.swing** y **java.awt**; en especial de las clases **Component**, **Container** y **JComponent**, que definen características comunes para la mayoría de los componentes Swing.

- 29.2** El componente que escucha un evento dado deberá implementar la interfaz para escuchar eventos apropiada.
- 29.3** Utilizar clases separadas para manejar eventos GUI produce componentes de software más reutilizables, confiables y legibles, los cuales pueden colocarse en paquetes y utilizarse en muchos programas.
- 29.4** Las ventanas son un recurso valioso del sistema, por lo que deben regresársele cuando ya no se les necesite.

EJERCICIOS DE AUTOEVALUACIÓN

- 29.1** Complete los espacios en blanco:
- El método _____ es llamado cuando el ratón se mueve y un componente que escucha eventos está registrado para manejar el evento.
 - El texto que no puede ser modificado por el usuario se llama texto _____.
 - Un _____ ordena los componentes GUI en un objeto **Container**.
 - El método **add** para adjuntar componentes GUI es un método de la clase _____.
 - GUI es un acrónimo de _____.
 - El método _____ se utiliza para establecer el administrador de diseño para un contenedor.
 - Una llamada al método **mouseDragged** va después de una llamada al método _____ y antes de una llamada al método _____.
- 29.2** Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- BorderLayout** es el administrador de diseño predeterminado para un panel de contenido.
 - Cuando el cursor del ratón se mueve hacia los límites de un componente GUI, se hace una llamada al método **mouseOver**.
 - Un objeto **JPanel** no puede agregarse a otro **JPanel**.
 - En un diseño **BorderLayout**, dos botones que se agreguen a la región **NORTH** aparecerán uno al lado del otro.
 - Cuando se utiliza **BorderLayout**, puede usarse un máximo de cinco componentes.
- 29.3** Encuentre el (los) error(es) en cada una de las siguientes instrucciones y explique cómo corregirlo(s).
- ```
a) nombreBoton = JButton("Leyenda");
b) JLabel unaEtiqueta, JLabel; // crea referencias
c) campoTexto = new JTextField(50, "Texto predeterminado");
d) Container c = getContentPane();
 setLayout(new BorderLayout());
 boton1 = new JButton("Estrella del norte");
 boton2 = new JButton("Polo sur");
 c.add(boton1);
 c.add(boton2);
```

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

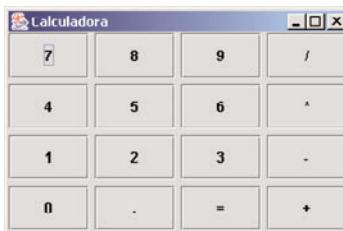
- 29.1** a) **mouseMoved**. b) No editable (de sólo lectura). c) Administrador de diseño. d) **Container**. e) Interfaz gráfica de usuario. f) **setLayout**. g) **mousePressed**, **mouseReleased**.
- 29.2** a) Verdadero.  
b) Falso. Se hace una llamada al método **mouseEntered**.  
c) Falso. Un **JPanel** puede agregarse a otro **JPanel**, ya que **JPanel** es una subclase indirecta de **Component**. Por lo tanto, un **JPanel** es un **Component**. Cualquier **Component** puede agregarse a un **Container**.  
d) Falso. Sólo se desplegará el último botón que se agregue. Recuerde que sólo debe agregarse un componente a cada región de un diseño **BorderLayout**.  
e) Verdadero.
- 29.3** a) se necesita **new** para crear un objeto.  
b) **JLabel** es el nombre de una clase y no puede utilizarse como nombre de variable.  
c) Los argumentos que se pasan al constructor están invertidos. El objeto **String** debe pasarse primero.  
d) Se ha establecido **BorderLayout** y los componentes se agregarán sin especificar la región, por lo que ambos se agregarán a la región central. Las instrucciones **add** apropiadas serían:  
**contenedor.add( boton1, BorderLayout.NORTH );**  
**contenedor.add( boton2, BorderLayout.SOUTH );**

## EJERCICIOS

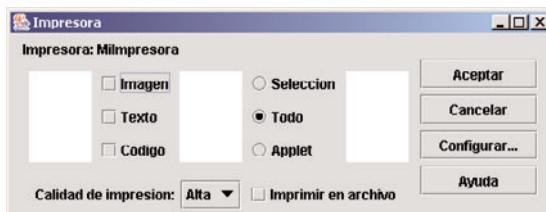
- 29.4** Complete los espacios en blanco:
- La clase **JTextField** hereda directamente de \_\_\_\_\_.
  - Los administradores de diseño que describimos en este capítulo son \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
  - El método \_\_\_\_\_ de **Container** adjunta un componente GUI a un contenedor.
  - El método \_\_\_\_\_ es llamado cuando se suelta uno de los botones del ratón (sin mover el ratón).
- 29.5** Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- Sólo puede usarse un administrador de diseño por cada objeto **Container**.
  - En un diseño **BorderLayout**, los componentes GUI pueden agregarse a un **Container** en cualquier orden.
  - El método **setFont** de **Graphics** se utiliza para establecer la fuente de los campos de texto.
  - Un objeto **Mouse** contiene un método llamado **mouseDragged**.
- 29.6** Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- Un objeto **JApplet** no tiene panel de contenido.
  - Un objeto **JPanel** es un objeto **JComponent**.
  - Un objeto **JPanel** es un objeto **Component**.
  - Un objeto **JLabel** es un objeto **Container**.
  - Un objeto **AbstractButton** es un objeto  **JButton**.
  - Un objeto **JTextField** es un objeto **Object**.
- 29.7** Encuentre los errores en cada una de las siguientes líneas de código y explique cómo corregirlos.
- `import javax.swing.* // incluye el paquete swing`
  - `objetoPanel.setLayout( 8, 8 ); // establece el diseño GridLayout`
  - `c.setLayout( new FlowLayout( FlowLayout.DEFAULT ) );`
  - `d.add( botonEste, EAST );// BorderLayout`
- 29.8** Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



- 29.9** Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



- 29.10** Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



- 29.11** Escriba un programa de conversión de temperatura, que convierta grados Fahrenheit a Centígrados. La temperatura en grados Fahrenheit deberá introducirse desde el teclado (mediante un objeto **JTextField**). Debe usarse un objeto **JLabel** para mostrar la temperatura convertida. Use la siguiente fórmula para la conversión:

$$\text{Centígrados} = \frac{5}{9} \times (\text{Fahrenheit} - 32)$$

- 29.12** Escriba una aplicación que permita al usuario dibujar un rectángulo, arrastrando el ratón en la ventana de aplicación. La coordenada superior izquierda deberá ser la ubicación en donde el usuario oprima el botón del ratón, y la coordenada inferior derecha deberá ser la ubicación en donde el usuario suelte el botón del ratón. Además, muestre el área del rectángulo en un **JLabel**, en la región **SOUTH** de un diseño **BorderLayout**. Todo el proceso de dibujo deberá realizarse en una subclase de **JPanel**. Use la siguiente fórmula para el área:

**área = ancho x altura**

- 29.13** Escriba un programa que muestre un círculo de tamaño aleatorio, que calcule y muestre el área, el radio, el diámetro y la circunferencia. Use las siguientes ecuaciones:  $diámetro = 2 \times radio$ ,  $área = \pi \times radio^2$ ,  $circunferencia = 2 \times \pi \times radio$ . Use la constante **Math.PI** para pi ( $\pi$ ). Todos los dibujos deberán realizarse en una subclase de **JPanel** y los resultados de los cálculos deberán mostrarse en un objeto **JTextArea** de sólo lectura.
- 29.14** Escriba un programa que utilice instrucciones **System.out.println** para imprimir los eventos según ocurran. Proporcione un objeto **JComboBox** con un mínimo de cuatro elementos. El usuario deberá ser capaz de seleccionar del objeto **JComboBox** un evento a “vigilar”. Cuando ocurra ese evento específico, muestre información acerca de él en un cuadro de diálogo de mensaje. Use el método **toString** en el objeto evento para convertirlo en una representación de cadena.
- 29.15** Escriba un programa utilizando métodos de la interfaz **MouseListener**, que permita al usuario oprimir el botón del ratón, arrastrar el ratón y soltar el botón del ratón. Cuando se suelte el botón del ratón, dibuje un rectángulo con la esquina superior izquierda, el ancho y la altura adecuados. [Pista: El método **mousePressed** debe capturar el conjunto de coordenadas en donde el usuario oprime inicialmente el botón del ratón y lo mantiene así, y el método **mouseReleased** debe capturar el conjunto de coordenadas en donde el usuario suelta el botón del ratón. Ambos métodos deberán almacenar los valores de coordenada apropiados. Todos los dibujos deberán realizarse en una subclase de **JPanel**, y todos los cálculos del ancho, la altura y la esquina superior izquierda deben realizarse mediante el método **paintComponent**, antes de que se dibuje la figura.]
- 29.16** Modifique el ejercicio 29.15 para proporcionar un efecto de “banda de hule”. Conforme el usuario arrastre el ratón, deberá poder ver el tamaño actual del rectángulo para saber exactamente cómo se verá el rectángulo cuando suelte el botón del ratón. [Pista: El método **mouseDragged** debe realizar las mismas tareas que **mouseReleased**.]
- 29.17** Modifique el ejercicio 29.16 para permitir al usuario seleccionar cuál figura dibujar. Un objeto **JComboBox** debe proporcionar opciones que incluyan, cuando menos, rectángulo, óvalo, línea y rectángulo redondeado.
- 29.18** Modifique el ejercicio 29.17 para permitir al usuario seleccionar el color de dibujo desde un cuadro de diálogo **JColorChooser**.
- 29.19** Modifique el ejercicio 29.18 para permitir al usuario especificar si una figura debe llenarse o vaciarse cuando ésta se dibuja. El usuario deberá hacer clic en un objeto **JCheckBox** para indicar si está llena o vacía.
- 29.20** (Aplicación de dibujo completa.) Por medio de las técnicas desarrolladas en los ejercicios 29.12 a 29.19, cree un programa de dibujo completo. Este programa debe utilizar los componentes GUI que vimos en este capítulo para permitir al usuario seleccionar la figura, el color y las características de relleno. Para este programa, cree sus propias clases (al igual que las de la jerarquía de clases que describimos en el ejercicio 27.19), a partir de las cuales se crearán objetos para guardar cada figura que dibuje el usuario. Las clases deberán almacenar la ubicación, las dimensiones y el color de cada figura, y deberán indicar si está llena o vacía. Sus clases deben derivarse de una clase llamada **MiFigura** que tenga todas las características comunes de cada tipo de figura. Cada subclase de **MiFigura** debe tener su propio método **draw**, el cual deberá devolver **void** y recibir un objeto **Graphics** como su argumento. Cree una subclase de **JPanel** llamada **PanelDibujo** para dibujar las figuras. Al llamar al método **paintComponent** de **PanelDibujo**, éste deberá recorrer el arreglo de figuras y mostrar cada una de ellas mediante una llamada polimórfica al método **draw** de la figura (con el objeto **Graphics** como argumento). El método **draw** de cada figura debe saber cómo dibujar la figura. Como mínimo, su programa debe proporcionar las siguientes clases: **MiLinea**, **MiOvalo**, **MiRectangulo**, **MiRectanguloRedondeado**. Diseñe la jerarquía de clases para obtener una máxima reutilización del código, y coloque todas sus clases en el paquete **figuras**. Importe este paquete en su programa. Cada figura debe almacenarse en un arreglo de objetos **MiFigura**, en donde **MiFigura** será la superclase en su jerarquía de clases de figuras (vea el ejercicio 27.19).
- 29.21** Modifique el ejercicio 29.20 para proporcionar un botón **Deshacer** que pueda utilizarse varias veces para deshacer la última operación de dibujo. Si no hay figuras en el arreglo de figuras, el botón **Deshacer** debe estar deshabilitado.



# 30

## Multimedia en Java: Imágenes, animación y audio

### Objetivos

- Comprender cómo obtener y desplegar imágenes.
- Crear animaciones a partir de secuencias de imágenes; controlar la velocidad y el parpadeo de animación.
- Obtener, reproducir, repetir y detener sonidos.
- Dar seguimiento a la carga de imágenes con la clase **MediaTracker**; crear mapas de imágenes.
- Personalizar los **applets** con la etiqueta **param**.

*La llanta que más rechina al rodar es la que obtiene el aceite.*

John Billings (Henry Wheeler Shaw)

*El ruido no demuestra nada. Con frecuencia, una gallina que tan solo pone un huevo, cacarea como si hubiera puesto un asteroide.*

Mark Twain

*Utilizaremos una señal que ya he utilizado, que se reconoce a lo lejos y es fácil de gritar. ¡Waa-huuu!*

Zane Grey

*Una pantalla grande solamente hace que una mala película sea doblemente mala.*

Samuel Goldwyn

*Entre el movimiento y el acto existe la sombra.*

Thomas Stearns Eliot

*Lo que experimentamos de la naturaleza es con modelos, y todos los modelos de la naturaleza son muy hermosos.*

Richard Buckminster Fuller



## Plan general

- 30.1 Introducción**
- 30.2 Cómo cargar, desplegar y escalar imágenes**
- 30.3 Cómo cargar y reproducir clips de audio**
- 30.4 Cómo animar una serie de imágenes**
- 30.5 Tópicos de animación**
- 30.6 Cómo personalizar applets por medio de la etiqueta `param` de HTML**
- 30.7 Mapas de imágenes**
- 30.8 Recursos en Internet y en la World Wide Web**

*Resumen • Terminología • Buenas prácticas de programación • Observaciones de apariencia visual • Tips de rendimiento • Tip de portabilidad • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios*

## 30.1 Introducción

Bienvenido a lo que probablemente representa la mayor revolución en la historia de la industria de la computación. Aquellos de nosotros que entramos al gremio hace algunas décadas estábamos interesados primordialmente en el uso de las computadoras para hacer cálculos numéricos a gran velocidad. Pero conforme evoluciona el campo de las computadoras, comenzamos a darnos cuenta de que en la actualidad también es igualmente importante la manipulación de datos. La “chispa” de Java es la *multimedia*, el uso de *sonido, imagen, gráficos y vídeo* para hacer que las aplicaciones “cobren vida”. En la actualidad, mucha gente considera al video en color de dos dimensiones como lo “último” en multimedia. Pero dentro de una década, esperamos toda clase de aplicaciones novedosas y excitantes en tres dimensiones. La programación multimedia ofrece muchos retos nuevos. El campo ya es enorme y crecerá rápidamente.

La gente se está apresurando para equipar a sus computadoras con multimedia. La mayoría de las computadoras nuevas se venden “listas para multimedia” con dispositivos de CD y DVD, tarjetas de sonido y, algunas veces, con capacidades especiales de vídeo.

Entre los usuarios que desean gráficos, los de dos dimensiones ya no son suficientes. Ahora mucha gente quiere gráficos en tres dimensiones, de alta resolución y en color. Las imágenes reales en tres dimensiones estarán disponibles a lo largo de la siguiente década. Imagine tener televisión de ultra alta resolución, con “teatro circular” y en tres dimensiones. Los eventos deportivos y de entretenimiento ¡tendrán lugar en su propia habitación! Los estudiantes de medicina alrededor del mundo verán las operaciones que se realizan a miles de kilómetros, como si ocurrieran en la misma habitación. La gente será capaz de aprender en sus casas a conducir por medio de simuladores extremadamente realistas, antes de colocarse frente al volante. Las posibilidades son excitantes e interminables.

La multimedia exige un extraordinario poder de cómputo. Hasta hace muy poco, las computadoras con este tipo de potencia no estaban disponibles. Pero los procesadores ultrarrápidos actuales como el SPARC Ultra de Sun Microsystems, el Pentium de Intel, el Alpha de Compaq Computer Corporation y el R8000 de MIPS/Silicon Graphics (entre otros) están haciendo posible la multimedia. Las industrias de cómputo y de comunicaciones serán las principales beneficiarias de la revolución de la multimedia. Los usuarios estarán dispuestos a pagar por procesadores más rápidos, más memoria y anchos de banda más grandes que se necesitarán para soportar las aplicaciones multimedia. Irónicamente, es probable que los usuarios no tengan que pagar más, ya que la ferocia competencia de estas industrias hace que los precios bajen.

Necesitamos lenguajes de programación para hacer más fácil la creación de aplicaciones multimedia. La mayoría de los lenguajes de programación no tienen incluidas las capacidades multimedia. Pero Java, a través de los paquetes de clases que son parte integral del mundo de la programación en Java, proporciona facilidades extendidas para multimedia que le permitirán comenzar a desarrollar de inmediato poderosas aplicaciones multimedia.

En este capítulo explicaremos una serie de ejemplos de “código vivo” que cubren muchas de las características multimedia que necesitará para construir aplicaciones útiles. Explicaremos los fundamentos de la manipula-

ción de imágenes, la creación de animaciones suaves, la reproducción de sonidos, la reproducción de vídeos, la creación de mapas de imágenes que pueden sentir cuando el apuntador se encuentra sobre ellos incluso sin un clic del ratón, y cómo personalizar los applets mediante los parámetros suministrados desde el archivo HTML que invoca al applet. Los ejercicios del capítulo sugieren proyectos interesantes y desafiantes, e incluso mencionan algunas ideas valiosas ¡que le podrían ayudar a hacer una fortuna! Cuando creamos estos ejercicios, las ideas seguían fluyendo. Con certeza, la multimedia promoverá la creatividad de formas que no hemos experimentado con las capacidades de cómputo “convencionales”.

## 30.2 Cómo cargar, desplegar y escalar imágenes

Las capacidades multimedia de Java incluyen gráficos, imágenes, animaciones, sonidos y vídeo. Comenzaremos nuestra explicación de multimedia con las imágenes.

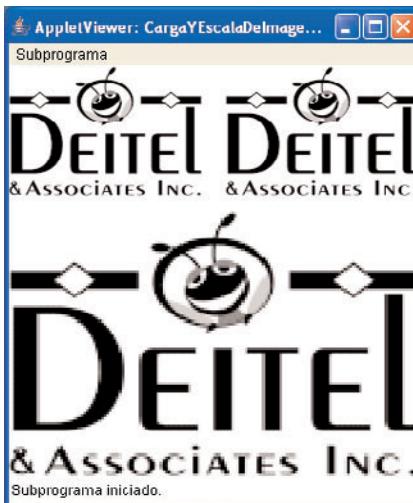
El applet de la figura 30.1 muestra cómo cargar una **Imagen** (**Image** del paquete **java.awt**) y cómo cargar un **ImageIcon** (del paquete **javax.swing**). El applet despliega la **Imagen** en su tamaño original y con una escala al doble de su longitud y altura original mediante dos versiones del método **drawImage** de **Graphics**. Además, el applet dibuja el **ImageIcon** mediante el método **paintIcon**. La clase **ImageIcon** es particularmente útil debido a que se puede utilizar para cargar fácilmente una imagen dentro de un applet o una aplicación.

---

```
1 // Figura 30.1: CargaYEscalaDeImagen.java
2 // Carga una imagen y la despliega en su tamaño original
3 // y la escala al doble de su ancho y altura.
4 // Carga y despliega la misma imagen como un IconoImagen.
5 import java.applet.Applet;
6 import java.awt.*;
7 import javax.swing.*;
8
9 public class CargaYEscalaDeImagen extends JApplet {
10 private Image logo1;
11 private ImageIcon logo2;
12
13 // carga la imagen cuando se carga el applet
14 public void init()
15 {
16 logo1 = getImage(getDocumentBase(), "logo.gif");
17 logo2 = new ImageIcon("logo.gif");
18 } // end method init
19
20 // despliega la imagen
21 public void paint(Graphics g)
22 {
23 // dibuja la imagen original
24 g.drawImage(logo1, 0, 0, this);
25
26 // dibuja la imagen escalada para que coincida con el ancho del applet
27 // y con la altura del applet menos 120 pixeles
28 g.drawImage(logo1, 0, 120,
29 getWidth(), getHeight() - 120, this);
30
31 // dibuja el icono utilizando su método paintIcon
32 logo2.paintIcon(this, g, 180, 0);
33 } // fin del método paint
34 } // fin de la clase CargaYEscalaDeImagen
```

---

Figura 30.1 Cómo cargar y desplegar una imagen dentro de un applet. (Parte 1 de 2.)



**Figura 30.1** Cómo cargar y desplegar una imagen dentro de un applet. (Parte 2 de 2.)

Las líneas 10 y 11 declaran una referencia a `Image` y una referencia a `ImageIcon`, respectivamente. La clase `Image` es una clase **abstract**; por lo tanto, usted no puede crear un objeto directamente de la clase `Image`. En vez de eso, debe pedir que se cargue y se le devuelva una `Image`. La clase `Applet` (la superclase de `JApplet`) proporciona un método que hace precisamente eso. La línea 16 en el método `init` del applet

```
logo1 = getImage(getDocumentBase(), "logo.gif");
```

utiliza el método `getImage` de `Applet` para cargar una `Imagen` dentro del applet. La versión de `getImage` toma dos argumentos, la ubicación en donde se almacena la imagen y el nombre del archivo de la imagen. En el primer argumento utilizamos el método `getDocumentBase` de `Applet` para determinar la ubicación de la imagen en Internet (o en su computadora si es de ahí de donde proviene). Asumimos que la imagen que se va a cargar se almacena en el mismo directorio que el archivo HTML que invoca al applet. El método `getDocumentBase` devuelve la ubicación del archivo HTML en Internet como un objeto de la clase `URL` (del paquete `java.net`). Una `URL` almacena un *Localizador Uniforme (o Universal) de Recursos*; un formato estándar para una dirección de una pieza de información en Internet. El segundo argumento especifica el nombre del archivo de la imagen. Actualmente Java soporta dos formatos de imagen, el **GIF** (*Formato de Intercambio de Gráficos*) y el **JPEG** (*Grupo unido de expertos en fotografía*). Los nombres de archivos para cada tipo terminan con `.gif` o `.jpg` (o `.jpeg`) respectivamente.

#### Tip de portabilidad 30.1



La clase `Image` es una clase **abstract**, por lo que no pueden crearse objetos de `Image` de manera directa. Para lograr la independencia de la plataforma, la implementación de Java en cada plataforma proporciona su propia subclase de `Image` para almacenar la información de la imagen.

Cuando se invoca al método `getImage`, se lanza un subprocesso de ejecución separado en el que se carga la imagen (o se descarga desde Internet). Esto permite al programa continuar la ejecución mientras se carga la imagen. [Nota: Si el archivo requerido no está disponible, el método `getImage` no indica un error.]

La clase `ImageIcon` no es una clase **abstract**; por lo tanto, usted puede crear un objeto a partir de `ImageIcon`. La línea 17 del método `init` del applet,

```
logo2 = new ImageIcon("logo.gif");
```

crea un objeto de `ImageIcon` que carga la misma imagen `logo.gif`. La clase `ImageIcon` proporciona muchos constructores que permiten inicializar con una imagen a un objeto `ImageIcon` desde la computadora local, o con una imagen almacenada en el servidor Web en Internet. La línea 24

```
g.drawImage(logo1, 0, 0, this);
```

utiliza el método `drawImage` de `Graphics`, el cual recibe cuatro argumentos (en realidad existen seis versiones sobrecargadas de este método). El primer argumento es una referencia al objeto `Image` en el que se almacena la imagen (`logo1`). El segundo y el tercer argumentos son las coordenadas *x* y *y* en donde debe desplegarse la imagen sobre el applet (las coordenadas indican la esquina superior izquierda de la imagen). El último argumento es una referencia a un objeto `ImageObserver`. Por lo general, el `ImageObserver` es un objeto sobre el que se despliega la imagen; utilizamos `this` para indicar el applet. Un `imageObserver` puede ser cualquier objeto que implemente la interfaz `ImageObserver`. La interfaz `ImageObserver` se implementa mediante la clase `Component` (una de las superclases indirectas de `Applet`). De esta manera, todos los `Component` pueden ser `ImageObserver`. Este argumento es importante cuando se despliegan imágenes de gran tamaño que requieren mucho tiempo para descargarse desde Internet. Es posible que un programa despliegue la imagen antes de que se complete la descarga. Al `ImageObserver` se le notifica automáticamente para que actualice la imagen que se desplegó, mientras se carga el resto de la imagen. Cuando ejecute este applet, observe con atención cómo se despliegan las piezas de la imagen mientras ésta se carga. [Nota: En las computadoras más rápidas, podría no notarse este efecto.]

Las líneas 28 y 29

```
g.drawImage(logo1, 0, 120,
 getWidth(), getHeight() - 120, this);
```

utilizan otra versión del método `drawImage` de `Graphics` para desplegar una versión *a escala* de la imagen. El cuarto y quinto argumentos especifican la *longitud* y la *altura* de la imagen para propósitos del desplegado. La imagen se escala automáticamente para que coincida con la longitud y la altura especificadas. El cuarto argumento indica que la longitud de la imagen a escala debe ser la longitud del applet y el quinto argumento indica que la altura debe ser de 120 pixeles menor que la altura del applet. La longitud y la altura del applet se determinan con los métodos `getWidth` y `getHeight` (que se heredan de la clase `Component`).

La línea 32

```
logo2.paintIcon(this, g, 180, 0);
```

utiliza el método `paintIcon` de `ImageIcon` para desplegar la imagen. El método requiere cuatro argumentos, una referencia al `Component` en el que se desplegará la imagen, una referencia al objeto, una referencia al objeto `Graphics` que se utilizará para modelar la imagen, y las coordenadas *x* y *y* de la esquina superior izquierda de la imagen.

Si compara las dos formas en las que cargamos y desplegamos las imágenes en este ejemplo, podrá ver que utilizar `ImageIcon` es más sencillo. Usted puede crear directamente objetos de la clase `ImageIcon` y no necesita utilizar una referencia a `ImageObserver` cuando despliega la imagen. Por esta razón, utilizaremos la clase `ImageIcon` en el resto del capítulo. [Nota: El método `paintIcon` de la clase `ImageIcon` no permite el escalamiento de una imagen. Sin embargo, la clase proporciona el método `getImage`, el cual devuelve una referencia a `Image` que puede utilizarse con el método `drawImage` de `Graphics` para desplegar la imagen seleccionada.]

### 30.3 Cómo cargar y reproducir clips de audio

Los programas en Java pueden manipular y reproducir *clips de audio*. Para los usuarios es fácil capturar sus propios clips de audio, y existe una gran variedad de clips que están disponibles en los productos de software y en Internet. Su sistema necesita estar equipado con el hardware para audio (bocinas y una tarjeta de sonido) para que sea capaz de reproducir clips de audio.

Java proporciona dos mecanismos para la reproducción de sonidos dentro de un applet, el método `play` de `Applet` y el método `play` de la interfaz `AudioClip`. Si usted quisiera reproducir un sonido una vez en un programa, el método `play` de `Applet` cargará el sonido y lo reproducirá una sola vez; el sonido se marca para el recolector de basura cuando termina la reproducción. El método `play` de `Applet` tiene dos formatos:

```
public void play(ubicación URL, Cadena nombreArchivoAudio);
public void play(URL URLaudio);
```

La primera versión carga el clip de audio almacenado en el archivo `nombreArchivoAudio` desde la `ubicación URL`, y reproduce el sonido. Por lo general, el primer argumento es una llamada al método `getDo-`

`cumentBase` o `getCodeBase`. El método `getDocumentBase` indica la ubicación del archivo HTML que cargó al applet. El método `getCodeBase` indica la ubicación del archivo `.class` del applet. La segunda versión del método `play` toma una `URL` que contiene la ubicación y el nombre del archivo del clip de audio. La instrucción

```
play(getCodeBase(), "hola.au");
```

carga el clip de audio en el archivo `hola.au` y lo reproduce una vez.

El *motor de audio* que reproduce los clips de audio soporta distintos formatos de archivos de sonido que incluyen el *formato de archivo de sonido de Sun* (extensión `.au`), el *formato de archivo Wave de Windows* (extensión `.wav`), el *formato de archivo AIFF de Macintosh* (extensiones `.aif` o `.aiff`) y el *formato de archivo Musical Instrument Digital Interface (MIDI)* (extensiones `.mid` o `.rmi`).

La figura 30.2 muestra la carga y la reproducción de un `AudioClip` (del paquete `java.applet`). Esta técnica es más flexible que el método `play` de `Applet`, ya que permite almacenar el sonido en el programa, de manera que se pueda reutilizar a lo largo de la ejecución del programa. El método `getAudioClip` de `Audio` tiene dos formas que toman los mismos argumentos que el método `play` descrito anteriormente. El método `getAudioClip` devuelve una referencia a un `AudioClip`. Una vez que se carga `AudioClip`, se pueden invocar tres métodos para el objeto: `play`, `loop` y `stop`. El método `play` reproduce el sonido solamente una vez. El método `loop` ejecuta repetidamente un clip de audio de fondo. El método `stop` termina el clip de audio que se encuentra en reproducción. En el programa, cada uno de estos métodos se asocia con un botón del applet.

---

```

1 // Figura 30.2: CargaYReproduccionDeAudio.java
2 // Carga un clip de audio y lo reproduce.
3 import java.applet.*;
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.swing.*;
7
8 public class CargaYReproduccionDeAudio extends JApplet {
9 private AudioClip sonido1, sonido2, sonidoActual;
10 private JButton reproduceSonido, repiteSonido, detieneSonido;
11 private JComboBox eligeSonido;
12
13 // carga la imagen cuando el applet comienza su ejecución
14 public void init()
15 {
16 Container c = getContentPane();
17 c.setLayout(new FlowLayout());
18
19 String elecciones[] = { "Bienvenido", "Hola" };
20 eligeSonido = new JComboBox(elecciones);
21 eligeSonido.addItemListener(
22 new ItemListener() {
23 public void itemStateChanged(ItemEvent e)
24 {
25 sonidoActual.stop();
26
27 sonidoActual =
28 eligeSonido.getSelectedIndex() == 0 ?
29 sonido1 : sonido2;
30 } // fin del método itemStateChanged
31 } // fin de la clase interna anónima
32); // fin de addItemListener

```

---

Figura 30.2 Cómo cargar y reproducir un `AudioClip`. (Parte 1 de 2.)

```

33 c.add(eligeSonido);
34
35 ButtonHandler manejador = new ButtonHandler();
36 reproduceSonido = new JButton("Reproducir");
37 reproduceSonido.addActionListener(manejador);
38 c.add(reproduceSonido);
39 repiteSonido = new JButton("Repetir");
40 repiteSonido.addActionListener(manejador);
41 c.add(repiteSonido);
42 detieneSonido = new JButton("Detener");
43 detieneSonido.addActionListener(manejador);
44 c.add(detieneSonido);
45
46 sonido1 = getAudioClip(
47 getDocumentBase(), "bienvenido.wav");
48 sonido2 = getAudioClip(
49 getDocumentBase(), "hola.au");
50 sonidoActual = sonido1;
51 } // fin del método init
52
53 // detiene el sonido cuando el usuario intercambia las páginas Web
54 // (es decir, sea amable con el usuario)
55 public void stop()
56 {
57 sonidoActual.stop();
58 } // fin del método stop
59
60 private class ButtonHandler implements ActionListener {
61 public void actionPerformed(ActionEvent e)
62 {
63 if (e.getSource() == reproduceSonido)
64 sonidoActual.play();
65 else if (e.getSource() == repiteSonido)
66 sonidoActual.loop();
67 else if (e.getSource() == detieneSonido)
68 sonidoActual.stop();
69 } // fin del método actionPerformed
70 } // fin de la clase interna ButtonHandler
71 } // fin de la clase CargaYReproduccionDeAudio

```



**Figura 30.2** Cómo cargar y reproducir un **AudioClip**. (Parte 2 de 2.)

Las líneas 46 a 49 del método **init** del applet

```

sonido1 = getAudioClip(
 getDocumentBase(), "bienvenido.wav");
sonido2 = getAudioClip(
 getDocumentBase(), "hola.au");

```

utilizan **getAudioClip** para cargar dos archivos de sonido: un archivo Wave de Windows (**bienvenido.wav**) y un archivo de audio de Sun (**hola.au**). El usuario puede seleccionar el clip de sonido a reproducir

desde `JComboBox.chooseSound`. Observe que el método `stop` del applet se redefine en la línea 55. Cuando el usuario intercambia las páginas Web, se invoca al método `stop` del applet. Esta versión de `stop` garantiza que un sonido en reproducción se detenga. De lo contrario, el clip de sonido continuará ejecutándose como fondo. En realidad éste no es un problema, pero puede ser tedioso para el usuario si el clip de sonido se repite. El método `stop` se proporciona aquí como un detalle para el usuario.



### Buena práctica de programación 30.1

*Cuando reproduzca sonidos en un applet o en una aplicación, proporcione un mecanismo para que el usuario pueda deshabilitar el sonido.*

## 30.4 Cómo animar una serie de imágenes

El siguiente ejemplo muestra la animación de series de imágenes almacenadas en un arreglo. La aplicación utiliza las mismas técnicas para cargar y desplegar `ImageIcons` que aparecen en la figura 30.1. En las ediciones previas de este texto, utilizamos una serie de ejemplos de animación para mostrar distintas técnicas para suavizar una animación. Una de las técnicas clave involucra el concepto llamado *gráficos con doble buffer*. Sin embargo, debido a que las nuevas características de los componentes GUI de Swing ya implementan las técnicas de suavización, simplemente nos podemos concentrar en el concepto de la animación.

La animación que presentamos en la figura 30.3 está diseñada como una subclase de `JPanel` (llamada `AnimadorLogo`), de modo que se puede adjuntar a una ventana de aplicación o posiblemente a un `JApplet`. Además, la clase `AnimadorLogo` define un método `main` (definido en la línea 71) para ejecutar la animación como una aplicación. El método `main` define una instancia de la clase `JFrame` y adjunta un objeto `AnimadorLogo` al `JFrame` para desplegar la animación.

```

1 // Figura 30.3: AnimadorLogo.java
2 // Animación de una serie de imágenes
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class AnimadorLogo extends JPanel
8 implements ActionListener {
9 protected ImageIcon imagenes[];
10 protected int totalImagenes = 30,
11 imagenActual = 0,
12 retardoAnimacion = 50; // 50 milisegundos de retardo
13 protected Timer cronoAnimacion;
14
15 public AnimadorLogo()
16 {
17 setSize(getPreferredSize());
18
19 imagenes = new ImageIcon[totalImagenes];
20
21 for (int i = 0; i < imagenes.length; ++i)
22 imagenes[i] =
23 new ImageIcon("imagenes/deitel" + i + ".gif");
24
25 iniciaAnimacion();
26 } // fin del constructor AnimadorLogo
27
28 public void paintComponent(Graphics g)
29 {

```

Figura 30.3 Animación de una serie de imágenes. (Parte 1 de 3.)

```
30 super.paintComponent(g);
31
32 if (imagenes[imagenActual].getImageLoadStatus() ==
33 MediaTracker.COMPLETE) {
34 imagenes[imagenActual].paintIcon(this, g, 0, 0);
35 imagenActual = (imagenActual + 1) % totalImagenes;
36 }
37 } // fin del método paintComponent
38
39 public void actionPerformed(ActionEvent e)
40 {
41 repaint();
42 } // fin del método actionPerformed
43
44 public void iniciaAnimacion()
45 {
46 if (cronoAnimacion == null) {
47 imagenActual = 0;
48 cronoAnimacion = new Timer(retardoAnimacion, this);
49 cronoAnimacion.start();
50 }
51 else // continúa desde la última imagen desplegada
52 if (! cronoAnimacion.isRunning())
53 cronoAnimacion.restart();
54 } // fin del método iniciaAnimacion
55
56 public void terminaAnimacion()
57 {
58 cronoAnimacion.stop();
59 } // fin del método terminaAnimacion
60
61 public Dimension getMinimumSize()
62 {
63 return getPreferredSize();
64 } // fin del método getMinimumSize
65
66 public Dimension getPreferredSize()
67 {
68 return new Dimension(160, 80);
69 } // fin del método getPreferredSize
70
71 public static void main(String args[])
72 {
73 AnimadorLogo anim = new AnimadorLogo();
74
75 JFrame app = new JFrame("Prueba Animacion");
76 app.getContentPane().add(anim, BorderLayout.CENTER);
77
78 app.addWindowListener(
79 new WindowAdapter() {
80 public void windowClosing(WindowEvent e)
81 {
82 System.exit(0);
83 }
84 } // fin del método windowClosing
84 } // fin de la clase interna anónima
```

Figura 30.3 Animación de una serie de imágenes. (Parte 2 de 3.)

```

85); // fin del addWindowListener
86
87 // Las constantes 10 y 30 se utilizan abajo para establecer el tamaño de
88 // la ventana 10 pixeles más ancha que la animación y
89 // 30 pixeles más alta que la animación.
90 app.setSize(anim.getPreferredSize().width + 10,
91 anim.getPreferredSize().height + 30);
92 app.show();
93 } // fin de main
94 } // fin de la clase AnimadorLogo

```



**Figura 30.3** Animación de una serie de imágenes. (Parte 3 de 3.)

La clase **AnimadorLogo** carga un arreglo de **ImageIcons** en su constructor. Mientras se crea la instancia de cada objeto **ImageIcon** en la estructura **for** de la línea 21, el constructor **ImageIcon** carga una imagen para la animación (existen 30 imágenes en total) con la instrucción

```
imagenes[i] =
 new ImageIcon("imagenes/deitel" + i + ".gif");
```

El argumento utiliza la concatenación de cadenas para ensamblar el nombre del archivo a partir de las partes **"imagenes/deitel"**, **i** y **".gif"**. Cada una de las imágenes de la animación se encuentra en uno de los archivos **"deitel0.gif"** a **"deitel29.gif"**. El valor de la variable de control de la estructura **for** se utiliza para seleccionar una de las 30 imágenes.

#### Tip de rendimiento 30.1



Es más eficiente cargar los marcos de la animación como una imagen, que cargar cada imagen por separado (puedes utilizar un programa de dibujo para combinar los marcos de la animación dentro de la imagen). Si las imágenes se cargan desde la World Wide Web, cada imagen cargada requiere una conexión separada hacia el sitio en donde se almacenan las imágenes.

#### Tip de rendimiento 30.2



Cargar todos los marcos de la animación como una imagen grande podría obligar a su programa a esperar para empezar a desplegar la animación.

Después de cargar las imágenes, el constructor llama a **iniciaAnimacion** (definida en la línea 44) para comenzar la animación. La animación es controlada por una instancia de la clase **Timer** (del paquete **javax.swing**). Un objeto de la clase **Timer** genera **ActionEvents** en un intervalo fijo en milisegundos (por lo general especificado como un argumento del constructor **Timer**) y notifica a todos sus **ActionListeners** registrados que ocurrió un evento. Las líneas 46 a 50

```

if (cronoAnimacion == null) {
 imagenActual = 0;
 cronoAnimacion = new Timer(retardoAnimacion, this);
 cronoAnimacion.start();
}

```

determinan si la referencia **cronoAnimacion** de **Timer** es **null**. Si es así, **imagenActual** se establece en 0 para indicar que la animación debe comenzar con la imagen del primer elemento del arreglo **imagenes**. La línea 48 asigna un nuevo objeto **Timer** a **cronoAnimacion**. El constructor **Timer** recibe dos argumentos, el retardo en milisegundos (en este ejemplo, el **retardoAnimacion** es 50 en milisegundos) y el **ActionListener** que responderá al **ActionEvent** de **Timer** (**this** **AnimadorLogo** implementa el **ActionLister-**

ner de la línea 8). La línea 49 inicia el objeto **Timer**. Una vez iniciado, **cronoAnimacion** generará un **ActionEvent** cada 50 milisegundos en este ejemplo. Las líneas 51 a 53

```
else // continua desde la última imagen desplegada
 if (! cronoAnimacion.isRunning())
 cronoAnimacion.restart();
```

son para programas que pueden detener la animación y reiniciarla. Por ejemplo, para hacer de una animación “amigable para el navegador” en un applet, la animación debe detenerse cuando el usuario intercambia entre páginas Web. Si el usuario regresa a la página Web con la animación, es posible llamar al método **iniciaAnimacion** para reiniciar la animación. La condición **if** de la línea 52 utiliza el método **isRunning** de **Timer** para determinar si el **Timer** se está ejecutando actualmente (es decir, generando eventos). Si no se está ejecutando, la línea 53 llama al método **restart** de **Timer** para indicar que el **Timer** debe comenzar a generar eventos nuevamente.

En respuesta a cada uno de los eventos de **Timer** de este ejemplo, el método **actionPerformed** (línea 39) llama al método **repaint**. Esto programa una llamada al método **update** de **AnimadorLogo** (heredado desde la clase **JPanel**), el cual, a su vez, llama al método **paintComponent** de **AnimadorLogo** (línea 28). Recuerde que cualquier subclase de **JComponent** que realiza un dibujo debe hacerlo en su método **paintComponent**. Como mencionamos en el capítulo 29, la primera instrucción de cualquier método **paintComponent** debe ser una llamada al método **paintComponent** de la superclase, para garantizar que los componentes Swing se desplieguen correctamente. La condición **if** de las líneas 32 y 33

```
if (imagenes[imagenActual].getImageLoadStatus() ==
 MediaTracker.COMPLETE) {
```

utiliza el método **getImageLoadStatus** de **ImageIcon** para determinar si la imagen a desplegar está completamente cargada en memoria. Sólo las imágenes completas deben desplegarse, para hacer la animación tan suave como sea posible. Cuando la imagen está completamente cargada, el método regresa **MediaTracker.COMPLETE**. Un objeto de la clase **MediaTracker** (del paquete **java.awt**) es utilizado por la clase **ImageIcon** para dar seguimiento a la carga de una imagen.

Cuando se cargan imágenes en un programa, dichas imágenes pueden registrarse con un objeto de la clase **MediaTracker**, para permitir al programa determinar cuándo una imagen se carga completamente. La clase **MediaTracker** también proporciona la habilidad de esperar la carga de una o varias imágenes, antes de permitir al programa continuar, y determina si ocurrió un error durante la carga de una imagen. Nosotros no necesitamos crear un **MediaTracker** de manera directa en este ejemplo, ya que la clase **ImageIcon** lo hace por nosotros. Sin embargo, cuando utilice la clase **Image** (como muestra la figura 30.1), es probable que quiera su propio **MediaTracker**.

#### **Tip de rendimiento 30.3**



Algunas personas que tienen experiencia con objetos **MediaTracker** han reportado que éstos tienen un efecto que va en detrimento del rendimiento. Mantenga esto en mente, como un área que analizará si necesita poner a punto sus aplicaciones multimedia.

#### **Tip de rendimiento 30.4**



Utilizar el método **waitForAll** de **MediaTracker** para esperar a que todas las imágenes registradas se descarguen completamente puede resultar en un gran retraso una vez que el programa comienza la ejecución y hasta que las imágenes en realidad se despliegan. Entre más grandes sean las imágenes, mayor será el tiempo que el usuario tendrá que esperar. Utilice el método **waitForAll** sólo para esperar que un número pequeño de imágenes se desplieguen completamente.

Si la imagen está completamente cargada, las líneas 34 y 35,

```
imagenes[imagenActual].paintIcon(this, g, 0, 0);
imagenActual = (imagenActual + 1) % totalImagenes;
```

dibujan el **ImageIcon** en el elemento **imagenActual** del arreglo y prepare la siguiente imagen a desplegar incrementando en 1 a **currentImage**. Observe el cálculo del módulo para garantizar que el valor de **currentImage** se establezca en 0, cuando se incremente a más de 29 (el último subíndice de elementos del arreglo).

El método `stopAnimation` (línea 56), detiene la animación con la línea 58,

```
cronoAnimacion.stop();
```

la cual utiliza el método `stop` de `Timer` para indicar que el `Timer` debe detener la generación de eventos. Esto, a su vez, previene que `actionPerformed` llame a `repaint` para iniciar el dibujo de la siguiente imagen del arreglo.

#### Observación de ingeniería de software 30.1



Cuando genere una animación para utilizarla en un applet, proporcione un mecanismo para deshabilitarla cuando el usuario navegue una nueva página Web diferente a la página en la que el applet de la animación reside.

Los métodos `getMinimumSize` (línea 61) y `getPreferredSize` (línea 66) se redefinen para ayudar al administrador de diseño a determinar el tamaño adecuado para un `AnimadorLogo` en un diseño. En este ejemplo, las imágenes son de 160 pixeles de ancho y de 80 pixeles de alto, por lo que el método `getPreferredSize` devuelve un objeto `Dimension` que contiene 160 y 80. El método `getMinimumSize` simplemente llama a `getPreferredSize` (una práctica común de programación). En `main` (línea 71), observe que el tamaño de la ventana de la aplicación se establece (líneas 90 y 91) en el mejor ancho de la animación más 10 pixeles, y en la mejor altura de la animación más 30 pixeles. Esto se debe a que el ancho y la altura de una ventana especifican los bordes externos de la ventana, no del *área del cliente* de la ventana (el área en donde pueden adjuntarse los componentes GUI).

En este ejemplo, pudimos aprovechar las diversas características que ayudan a producir animaciones suaves y controlables; los objetos `ImageIcon` cargaron las imágenes, un objeto de una subclase de `JPanel` desplegó las imágenes, y un objeto `Timer` controló la animación.

## 30.5 Tópicos de animación

Cuando ejecute la aplicación de la figura 30.3, podrá observar que la imagen se lleva tiempo en cargar. Si una animación no se diseña correctamente, esto con frecuencia da como resultado que las imágenes se desplieguen parcialmente. Es posible que usted vea que cada imagen se despliega por partes. Con frecuencia, esto es el resultado del formato que se utiliza para la imagen. Por ejemplo, las imágenes GIF pueden almacenarse en formatos *entrelazados* y *no entrelazados*. El formato indica el orden en el que se almacenan los pixeles de la imagen. Los pixeles de una imagen no entrelazada se almacenan en el mismo orden en el que los pixeles aparecen en la pantalla. Conforme se despliega una imagen no entrelazada, ésta aparece en pedazos de arriba hacia abajo, conforme se lee la información sobre los pixeles. Los pixeles de una imagen entrelazada se almacenan en filas de pixeles, sin embargo, las filas están en desorden. Por ejemplo, las filas de pixeles de la imagen pueden almacenarse en el orden 1, 5, 9, 13, ..., seguido por 2, 6, 10, 14, ..., y así sucesivamente. Cuando la imagen se despliega, ésta parece desvanecida, ya que el primer lote de filas presenta una imagen borrosa, y los lotes subsiguientes de filas mejoran la imagen desplegada, hasta que la totalidad de la imagen se completa. Para ayudar a evitar que aparezcan imágenes parciales en versiones anteriores de Java, dimos seguimiento a la carga de imágenes por medio de un objeto `MediaTracker`. Sólo se despliegan imágenes totalmente cargadas, para producir la animación más suave. Cada imagen a rastrear debe registrarse con el `MediaTracker`. Esto ahora se lleva a cabo por medio del constructor de la clase `ImageIcon`.

#### Observación de ingeniería de software 30.2



La clase `ImageIcon` utiliza un objeto `MediaTracker` para determinar el estado de la imagen que está cargando.

#### Buena práctica de programación 30.2



En un applet, siempre despliegue algo mientras se cargan las imágenes. Entre más tiempo tenga que esperar un usuario para ver información en la pantalla, es más probable que abandone la página Web antes de que la información aparezca.

Otro problema común con las animaciones es que la animación *parpadea* conforme cada imagen se despliega. Esto se debe a que se llama al método `update` en respuesta a cada `repaint`. En componentes GUI

de AWT, cuando **update** limpia el fondo del componente GUI, lo hace dibujando un rectángulo del tamaño del componente, relleno con el color de fondo actual. Esto cubre la imagen que se acababa de desplegar. Por lo tanto, la animación dibujaría una imagen, dormiría por una fracción de segundo, limpiaría el fondo (ocasionando un parpadeo), y dibujaría la siguiente imagen. En subclases **JPanel** de Swing (o de cualquier otro componente Swing), el método **update** se redefine para evitar limpiar el fondo, si el componente es transparente (el fondo se limpiará si el componente es opaco). Esto ayuda a eliminar el parpadeo.



### Observación de apariencia visual 30.1

*Los componentes Swing redefinen el método **update** para evitar que se limpie el fondo (en el caso de componentes transparentes), en respuesta a mensajes **repaint**.*

Si desea desarrollar aplicaciones basadas en multimedia, sus usuarios querrán audio y animaciones suaves. Las presentaciones dispares son inaceptables. Esto con frecuencia ocurre cuando escribe aplicaciones que dibujan directamente en la pantalla. Otra técnica que se utiliza para producir animaciones suaves (y otros gráficos) es la de *gráficos con doble búfer*. Mientras el programa interpreta una imagen en la pantalla, éste puede construir la siguiente imagen en un *búfer fuera de pantalla*. Después, cuando es momento de que se despliegue la siguiente imagen, puede colocarla suavemente en la pantalla. Por supuesto, existe un *equilibrio espacio/tiempo*. La memoria adicional requerida puede ser importante, pero el rendimiento mejorado del despliegue lo vale.

Los gráficos con doble búfer también son útiles en programas que necesitan utilizar capacidades de dibujo en métodos diferentes de **paint** o **paintComponent** (en donde hemos hecho todos nuestros dibujos hasta este punto). El búfer fuera de pantalla puede pasarse entre métodos, o incluso entre objetos de diferentes clases, para permitir a otros métodos u objetos dibujar en el búfer fuera de pantalla. Los resultados del dibujo pueden entonces desplegarse en otro momento.



### Tip de rendimiento 30.5

*El doble búfer puede reducir o eliminar el parpadeo de una animación, pero puede disminuir visiblemente la velocidad a la que se ejecuta la animación.*

Cuando todos los pixeles de una imagen no se despliegan al mismo tiempo, una animación tiene más parpadeo. Cuando una imagen se dibuja por medio de gráficos con doble búfer, en el momento en que la imagen se despliega, ésta habrá sido dibujada fuera de la pantalla, y las imágenes parciales que el usuario normalmente vería, están ocultas para él. Todos los pixeles se desplegarán para el usuario en un “tris”, para que el parpadeo se vea substancialmente disminuido, o para que desaparezca.

Los conceptos básicos de un gráfico con doble búfer son los siguientes: crear una **Imagen** en blanco, dibujar en la **Imagen** en blanco (utilizando métodos de la clase **Graphics**) y desplegar la imagen. La **Imagen** almacena los pixeles que se copiarán en la pantalla. La referencia **Graphics** se utiliza para dibujar los pixeles. Toda imagen tiene un contexto gráfico asociado; es decir, un objeto de la clase **Graphics** que permite que el dibujo se realice. Las referencias **Image** y **Graphics** utilizadas para los gráficos con doble búfer con frecuencia se conocen como *imagen fuera de la pantalla* y *contexto gráfico fuera de la pantalla*, debido a que en realidad no manipulan pixeles de pantalla.

Los componentes GUI de Swing se despliegan utilizando las capacidades de dibujo de Java. Por lo tanto, los componentes GUI de Swing están sujetos a muchos de los mismos problemas que se encuentran en una animación típica. De manera predeterminada, Swing utiliza gráficos con doble búfer para interpretar todos los componentes GUI. Al diseñar nuestro **AnimadorLogo** como una subclase de **JPanel**, podemos aprovechar los gráficos con doble búfer integrados de Swing para producir las animaciones más suaves.



### Observación de apariencia visual 30.2

*Los componentes GUI de Swing se interpretan utilizando gráficos con doble búfer, de manera predeterminada.*

## 30.6 Cómo personalizar applets por medio de la etiqueta param de HTML

Cuando navegue en la World Wide Web, con frecuencia encontrará applets que son del dominio público; puede utilizarlos de manera gratuita en sus propias páginas Web (normalmente como un intercambio por los créditos del creador del applet). Una característica común de dichos applets es la capacidad de personalizar el applet a

través de los parámetros que se proporcionan en el archivo HTML que invoca el applet. Por ejemplo, el siguiente código HTML del archivo AppletLogo.html

```
<html>
<applet code="AppletLogo.class" width=400 height=400>
<param name="Imagenestotales" value="30">
<param name="nombreImagen" value="deitel">
<param name="retardoanimacion" value="200">
</applet>
</html>
```

invoca al applet **AppletLogo** (figura 30.4) y especifica tres parámetros. Las líneas de la *etiqueta param* deben aparecer entre las etiquetas **applet** inicial y final. Estos valores pueden entonces utilizarse para personalizar el applet. Cualquier número de etiquetas **param** puede aparecer entre las etiquetas **applet** inicial y final. Cada parámetro tiene un **nombre** y un **valor**. El método **getParameter** de **Applet** se utiliza para obtener el **valor** asociado con un parámetro específico y devuelve el **valor** como una **String**. El argumento pasado a **getParameter** es una **String** que contiene el nombre del parámetro en la etiqueta **param**. Por ejemplo, la instrucción

```
parametro = getParameter("retardoanimacion");
```

obtiene el valor asociado con el parámetro **retardoanimacion**, y lo asigna a la referencia **parametro** de **String**. Si no hay una etiqueta **param** que contenga el parámetro especificado, **getParameter** devuelve **null**.

---

```

1 // Figura 30.4: AnimadorLogo.java
2 // Animación de una serie de imágenes
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class AnimadorLogo extends JPanel
8 implements ActionListener {
9 protected ImageIcon imagenes[];
10 protected int totalImagenes = 30,
11 imagenActual = 0,
12 retardoImagen = 50; // retardo de 50 milisegundos
13 protected String nombreImagen = "deitel";
14 protected Timer cronoAnimacion;
15
16 public AnimadorLogo()
17 {
18 inicializaAnimacion();
19 } // fin del constructor AnimadorLogo
20
21 // constructor new para soportar la personalización
22 public AnimadorLogo(int num, int retardo, String nombre)
23 {
24 totalImagenes = num;
25 retardoImagen = retardo;
26 nombreImagen = nombre;
27
28 inicializaAnimacion();
29 } // fin del constructor AnimadorLogo
30
```

---

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **param** de HTML:  
**AnimadorLogo.java**. (Parte 1 de 3.)

```
31 private void inicializaAnimacion()
32 {
33 imagenes = new ImageIcon[totalImagenes];
34
35 for (int i = 0; i < imagenes.length; ++i)
36 imagenes[i] = new ImageIcon("imagenes/" +
37 nombreImagen + i + ".gif");
38
39 // se movió aquí para que getPreferredSize pueda verificar el tamaño de
40 // la primera imagen cargada.
41 setSize(getPreferredSize());
42
43 iniciaAnimacion();
44 } // fin del método inicializaAnimacion
45
46 public void paintComponent(Graphics g)
47 {
48 super.paintComponent(g);
49
50 if (imagenes[imagenActual].getImageLoadStatus() ==
51 MediaTracker.COMPLETE) {
52 imagenes[imagenActual].paintIcon(this, g, 0, 0);
53 imagenActual = (imagenActual + 1) % totalImagenes;
54 } // fin de if
55 } // fin del método paintComponent
56
57 public void actionPerformed(ActionEvent e)
58 {
59 repaint();
60 } // fin del método actionPerformed
61
62 public void iniciaAnimacion()
63 {
64 if (cronoAnimacion == null) {
65 imagenActual = 0;
66 cronoAnimacion = new Timer(retardoImagen, this);
67 cronoAnimacion.start();
68 }
69 else // continúa desde la última imagen desplegada
70 if (! cronoAnimacion.isRunning())
71 cronoAnimacion.restart();
72 } // fin del método iniciaAnimacion
73
74 public void detieneAnimacion()
75 {
76 cronoAnimacion.stop();
77 } // fin del método detieneAnimacion
78
79 public Dimension getMinimumSize()
80 {
81 return getPreferredSize();
82 } // fin del método getMinimumSize
83
```

Figura 30.4 Cómo personalizar un applet a través de la etiqueta **param** de HTML;  
**AnimadorLogo.java**. (Parte 2 de 3.)

---

```

84 public Dimension getPreferredSize()
85 {
86 return new Dimension(imagenes[0].getIconWidth(),
87 imagenes[0].getIconHeight());
88 } // fin del método getPreferredSize
89
90 public static void main(String args[])
91 {
92 AnimadorLogo anim = new AnimadorLogo();
93
94 JFrame app = new JFrame("Prueba Animacion");
95 app.getContentPane().add(anim, BorderLayout.CENTER);
96
97 app.addWindowListener(
98 new WindowAdapter() {
99 public void windowClosing(WindowEvent e)
100 {
101 System.exit(0);
102 } // fin del método windowClosing
103 } // y de la clase interna anónima
104); // y de addWindowListener
105
106 app.setSize(anim.getPreferredSize().width + 10,
107 anim.getPreferredSize().height + 30);
108 app.show();
109 } // fin de main
110 } // fin de la clase AnimadorLogo

```

---

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **param** de HTML;  
**AnimadorLogo.java**. (Parte 3 de 3.)

---

```

111 // Figura 30.4: AppletLogo.java
112 // Personalización de un applet por medio de parámetros en HTML
113 //
114 // El parámetro HTML "retardoAnimacion" es un int que indica
115 // los milisegundos de retardo entre las imágenes (50 de manera
116 // predeterminada).
117 // El parámetro HTML "nombreImagen" es el nombre de la base de las imágenes
118 // que se desplegará (es decir, "deitel" es el nombre base de las
119 // imágenes "deitel0.gif," "deitell.gif," etc.). El applet
120 // asume que las imágenes están en un subdirectorio "imagenes" del
121 // directorio en el cual reside el applet.
122 //
123 // El parámetro HTML "totalImagenes" es un entero que representa el
124 // número total
125 // de imágenes en la animación. El applet asume que las imágenes están
126 // numeradas desde 0 hasta totalImagenes - 1 (30 de manera predeterminada).
127 import java.awt.*;
128 import javax.swing.*;
129
130 public class AppletLogo extends JApplet{

```

---

**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **param** de HTML;  
**AppletLogo.java**. (Parte 1 de 2.)

```

131 public void init()
132 {
133 String parametro;
134
135 parametro = getParameter("retardoanimacion");
136 int retardoAnimacion = (parametro == null ? 50 :
137 Integer.parseInt(parametro));
138
139 String nombreImagen = getParameter("nombreImagen");
140
141 parametro = getParameter("totalImagenes");
142 int totalImagenes = (parametro == null ? 0 :
143 Integer.parseInt(parametro));
144
145 // Crea una instancia de AnimadorLogo
146 AnimadorLogo animador;
147
148 if (nombreImagen == null || totalImagenes == 0)
149 animador = new AnimadorLogo();
150 else
151 animador = new AnimadorLogo(totalImagenes,
152 retardoAnimacion, nombreImagen);
153
154 setSize(animador.getPreferredSize().width,
155 animador.getPreferredSize().height);
156 getContentPane().add(animador, BorderLayout.CENTER);
157
158 animador.iniciaAnimacion();
159 } // fin del método init
160 } // fin de la clase AppletLogo

```



**Figura 30.4** Cómo personalizar un applet a través de la etiqueta **param** de HTML; **AppletLogo.java**. (Parte 2 de 2.)

En la figura 30.4 modificamos la clase **AnimadorLogo** para poder utilizarla desde un applet y personalizarla a través de los parámetros del archivo HTML del applet. La clase **AppletLogo** permite a los diseñadores de páginas Web personalizar la animación para utilizarla en sus propias imágenes. Se proporcionan tres parámetros. El parámetro **retardoAnimacion** es el número de milisegundos a dormir entre las imágenes que se despliegan. Este valor se convertirá en un entero y se utilizará como el valor para la variable de instancia **sleepTime**. El parámetro **nombreImagen** es el nombre base de las imágenes a cargar. Esta **String** se asignará a la variable de instancia **nombreImagen**. El applet asume que las imágenes se encuentran en un subdirectorío llamado **imagenes** que puede localizarse en el mismo directorio del applet. El applet también asume que los nombres de los archivos de imágenes están numerados a partir de 0. El parámetro **totalImagenes** representa el número total de imágenes en la animación. Su valor se convertirá en un entero y se asignará a la variable de instancia **totalImagenes**.

La clase **AnimadorLogo** tiene diversas características nuevas para permitir su uso y su personalización en el **AppletLogo**. En la línea 13, se define la variable de instancia **nombreImagen**. Ésta almacenará el

nombre base predeterminado “**deitel**”, que es parte de todo nombre de archivo, o almacenará el nombre personalizado pasado al applet desde el documento HTML.

Ahora hay dos constructores; uno predeterminado (línea 16) y otro que toma argumentos para personalizar la animación (línea 22). Ambos constructores pueden llamar a nuestro nuevo método de utilidad **inicializaAnimacion** (línea 31) para cargar las imágenes e iniciar la animación. Las instrucciones de **inicializaAnimacion** estaban originalmente en el constructor predeterminado. La llamada al método **setSize** de la línea 41 (que se utiliza para preceder la carga de imágenes) se movió hacia la línea 41 para que el **AnimadorLogo** pudiera establecer un nuevo tamaño, de acuerdo con el ancho y el alto de la primera imagen de la animación. Para acomodar el nuevo tamaño basado en la primera imagen, el método **getPreferredSize** (línea 84) ahora devuelve un objeto **Dimension** que contiene el ancho y la altura de la primera imagen de la animación.

La clase **AppletLogo** (línea 130) define un método **init** en el que se leen los tres parámetros HTML con el método **getParameter** de **Applet** (líneas 135, 139 y 141). Después de que se leen los parámetros y de que los dos parámetros enteros se convierten en valores **int**, la estructura **if/else** de las líneas 148 a 152 crea un **AnimadorLogo**. Si el **nombreImagen** es **null**, o **totalImagenes** es 0, se llama al constructor predeterminado **AnimadorLogo** y se utilizará la animación predeterminada. De lo contrario, **totalImagenes**, **retardoAnimación** y **nombreImagen** se pasan al constructor de tres argumentos **AnimadorLogo**, y éste utiliza dichos argumentos para personalizar la animación.

## 30.7 Mapas de imágenes

Una técnica común para crear páginas Web interesantes es el uso de *mapas de imágenes*. Un mapa de imágenes es una imagen que tiene *áreas sensibles* en donde el usuario puede hacer clic para realizar una tarea como cargar una página Web diferente en un navegador. Cuando el usuario posiciona el puntero del ratón sobre un área sensible, normalmente se despliega un mensaje descriptivo en el área de estado del navegador. Esta técnica puede utilizarse para implementar un sistema de *ayuda de burbuja*. Cuando el usuario posiciona el puntero del ratón sobre un elemento en particular de la pantalla, un sistema con ayuda de burbuja normalmente despliega un mensaje en una pequeña ventana que aparece sobre el elemento de la pantalla. En Java, el mensaje puede desplegarse en la barra de estado.

La figura 30.5 carga una imagen que contiene diversos iconos del *Java Multimedia Cyber Classroom*, el CD interactivo con la versión multimedia de este texto. Estos iconos pueden parecerle conocidos; están diseñados para imitar los iconos que utilizamos en este libro. El programa permite al usuario posicionar el puntero del ratón sobre un ícono y desplegar un mensaje descriptivo para el ícono. El manejador de eventos **mouseMoved** (línea 24) toma la coordenada *x* del ratón y la pasa al método **translateLocation** (línea 42). La coordenada *x* se evalúa para determinar el ícono sobre el que se posicionó el ratón cuando se llamó al método **mouseMoved**. El método **translateLocation** entonces devuelve un mensaje que indica lo que el ícono representa. Este mensaje se despliega en la barra de estado del **appletviewer** (o del navegador).

---

```

1 // Figura 30.5: MapaImagen.java
2 // Demostración de un mapa de imágenes.
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class MapaImagen extends JApplet {
8 private ImageIcon mapaImagen;
9 private int ancho, alto;
10
11 public void init()
12 {
13 addMouseListener(
14 new MouseAdapter() {

```

---

**Figura 30.5** Demostración de un mapa de imágenes. (Parte 1 de 3.)

```

15 public void mouseExited(MouseEvent e)
16 {
17 showStatus("Apuntador fuera del applet");
18 } // fin de método mouseExited
19 } // fin de la clase interna anónima
20); // fin de addMouseListener
21
22 addMouseMotionListener(
23 new MouseMotionAdapter() {
24 public void mouseMoved(MouseEvent e)
25 {
26 showStatus(trasladaUbicacion(e.getX()));
27 } // fin de método mouseMoved
28 } // fin de la clase interna anónima
29); // fin de addMouseMotionListener
30
31 mapaImagen = new ImageIcon("iconos2.gif");
32 ancho = mapaImagen.getIconWidth();
33 alto = mapaImagen.getIconHeight();
34 setSize(ancho, alto);
35 } // fin del método init
36
37 public void paint(Graphics g)
38 {
39 mapaImagen.paintIcon(this, g, 0, 0);
40 } // fin del método paint
41
42 public String trasladaUbicacion(int x)
43 {
44 // determina el ancho de cada icono (existen 6)
45 int anchoIcono = ancho / 6;
46
47 if (x >= 0 && x <= anchoIcono)
48 return "Error comun de programacion";
49 else if (x > anchoIcono && x <= anchoIcono * 2)
50 return "Buena practica de programacion";
51 else if (x > anchoIcono * 2 && x <= anchoIcono * 3)
52 return "Tip de rendimiento";
53 else if (x > anchoIcono * 3 && x <= anchoIcono * 4)
54 return "Tip de portabilidad";
55 else if (x > anchoIcono * 4 && x <= anchoIcono * 5)
56 return "Observacion de ingenieria de software";
57 else if (x > anchoIcono * 5 && x <= anchoIcono * 6)
58 return "Tip para prueba y depuracion";
59
60 return "";
61 } // fin del método trasladaUbicacion
62 } // fin de la clase MapaImagen

```



**Figura 30.5** Demostración de un mapa de imágenes. (Parte 2 de 3.)



**Figura 30.5** Demostración de un mapa de imágenes. (Parte 3 de 3.)

Hacer clic en este applet no ocasionará acción alguna. Si fuéramos a agregar capacidades de red, podríamos modificar este applet para permitir que cada ícono estuviera asociado con una URL diferente.

## 30.8 Recursos en Internet y en la World Wide Web

Esta sección presenta diversos recursos en Internet y en la Web para sitios relacionados con multimedia.

<http://www.nasa.gov/gallery/index.html>

La galería multimedia de la NASA contiene una amplia variedad de imágenes, clips de audio y vídeo que puede descargar, para utilizarlos para probar sus programas multimedia en Java.

<http://sunsite.sut.ac.jp/multimed/>

La Sunsite Japan Multimedia Collection también proporciona una amplia variedad de imágenes, clips de audio y vídeo que puede descargar para fines educativos.

<http://www.anbg.gov.au/anbg/index.html>

El sitio Web Australian National Botanics Gardens proporciona vínculos hacia sonidos de muchos animales. Pruebe el vínculo *Common Birds*.

## RESUMEN

- El método `getImage` de `Applet` carga una `Imagen`. Una versión de `getImage` toma dos argumentos, una ubicación en donde se almacena el archivo y el nombre del archivo de la imagen.
- El método `getDocumentBase` de `Applet` devuelve la ubicación del archivo HTML del applet en Internet, como un objeto de la clase `URL` (del paquete `java.net`).
- Una `URL` almacena un Localizador Uniforme (o Universal) de Recursos; un formato estándar para una dirección de una pieza de información en Internet.
- Java soporta dos formatos de imagen, GIF (Formato de Intercambio de Gráficos) y JPEG (Grupo unido de expertos en fotografía). Los nombres de archivos para estos tipos terminan con `.gif` o `.jpg` (o `.jpeg`), respectivamente.
- La clase `ImageIcon` proporciona constructores que permiten a un objeto `ImageIcon` inicializarse con una imagen desde la computadora local, o con una imagen almacenada en un servidor Web en Internet.

- El método **Graphics** de **drawImage** recibe cuatro argumentos, una referencia al objeto **Image** en el cual se almacena la imagen, las coordenadas *x* y *y* en donde debe desplegarse la imagen y una referencia al objeto **ImageObserver**.
- Otra versión del método **drawImage** de **Graphics** despliega una imagen *a escala*. El cuarto y el quinto argumentos especifican el ancho y la altura de la imagen para propósitos del desplegado de dicha imagen.
- La interfaz **ImageObserver** se implementa mediante la clase **Component** (una superclase indirecta de **Applet**). A las **ImageObserver** se les notifica la actualización de una imagen que se despliega mientras se descarga el resto de la imagen.
- El método **paintIcon** de **ImageIcon** despliega la imagen de **ImageIcon**. El método requiere cuatro argumentos: una referencia al **Component** en el cual se desplegará la imagen, una referencia al objeto **Graphics** que se utiliza para interpretar la imagen, la coordenada *x* y *y* de la esquina superior izquierda de la imagen, y la coordenada *y* de la esquina superior izquierda de la imagen.
- El método **paintIcon** de la clase **ImageIcon** no permite escalar ninguna imagen. La clase proporciona el método **getImage** el cual devuelve una referencia a **Image** que puede utilizarse con el método **drawImage** de **Graphics** para desplegar una versión a escala de una imagen.
- El método **play** de **Applet** tiene dos formas:

```
public void play(URL ubicación, String nombreArchivoDeSonido);
public void play(URL URLdeSonido);
```

Una versión carga el clip de audio almacenado en el archivo **nombreArchivoDeSonido** desde la **ubicación** y reproduce el sonido. El otro toma una URL que contiene la ubicación y el nombre del archivo del clip de audio.

- El método **getDocumentBase** de **Applet** indica la ubicación del archivo HTML que cargó el applet. El método **getCodeBase** indica en dónde se localiza el archivo **.class** para el applet que se carga.
- El motor de audio que reproduce los clips de audio soporta varios formatos de audio que incluyen el formato de archivo de sonido de Sun (extensión **.au**), formato de archivo Wave de Windows (extensión **.wav**), el formato de archivo AIFF de Macintosh (extensión **.aif** o **.aiff**) y el formato de archivo Musical Instrument Digital Interface (MIDI) (extensión **.mid** o **.rmi**).
- El método **getAudioClip** de **Applet** tiene dos formas que toman los mismos argumentos que el método **play**. El método **getAudioClip** devuelve una referencia a un **AudioClip**. **AudioClip** tiene tres métodos, **play**, **loop** y **stop**. El método **play** reproduce una vez el sonido. El método **loop** repite de manera continua el clip de audio. El método **stop** termina un clip de audio que está en reproducción.
- Los objetos **Timer** generan **ActionEvents** en intervalos fijos en milisegundos y notifica a sus **ActionListeners** que ocurrieron los eventos. El constructor **Timer** recibe dos argumentos, el retardo en milisegundos y el **ActionListener**. El método **start** de **Timer** indica que **Timer** debe comenzar a generar eventos. El método **restart** de **Timer** indica que **Timer** debe comenzar nuevamente a generar eventos.
- El método **getImageLoadStatus** de **ImageIcon** determina si una imagen está cargada completamente en memoria. El método devuelve **MediaTracker.COMPLETE** si la imagen ya se cargó por completo.
- Las imágenes pueden registrarse con un objeto de la clase **MediaTracker** para permitir al programa determinar cuándo una imagen está cargada completamente.
- Las imágenes GIF pueden almacenarse en formatos entrelazados y no entrelazados. El formato indica el orden en el cual se almacenan los pixeles de la imagen. Mientras se despliega una imagen no entrelazada, los trozos de imagen aparecen de arriba hacia abajo mientras se lee la información de los pixeles. Los pixeles de una imagen entrelazada se almacenan en filas de pixeles, pero las filas están en desorden. Cuando se despliega la imagen, ésta parece desvanecida, ya que el primer lote de filas presenta una imagen borrosa, y los lotes subsiguientes de filas mejoran la imagen desplegada, hasta que la totalidad de la imagen se completa.
- Un problema común con las animaciones es que la animación parpadea al aparecer cada imagen. Por lo general, esto se debe a que se llama al método **update** en respuesta a cada **repaint**. En las subclases del **JPanel** de Swing (o cualquier otro componente de Swing), el método **update** se redefine para evitar la limpieza del fondo.
- Una técnica utilizada para producir animaciones suaves son los gráficos con doble búfer. Mientras el programa dibuja una imagen en la pantalla, puede construir la siguiente imagen en un búfer fuera de la pantalla. Entonces, cuando es tiempo de desplegar la siguiente imagen, ésta puede colocarse suavemente en la pantalla.
- Los componentes GUI de Swing se despliegan mediante el uso de las capacidades de dibujo de Swing. Por lo tanto, los componentes GUI de Swing están sujetos a muchos de los mismos problemas que se encuentran en una animación típica. De manera predeterminada, Swing utiliza doble búfer para interpretar todos los componentes GUI de Swing.

- Los applets pueden personalizarse mediante los parámetros (la etiqueta `<param>`) que se suministran en el archivo HTML que invoca al applet. Las líneas de la etiqueta `<param>` deben aparecer entre la etiqueta de `applet` de inicio y la etiqueta `applet` final. Cada parámetro tiene un **nombre** y un **valor**.
- El método `getParameter` de `Applet` obtiene el **valor** asociado con un parámetro específico y devuelve el **valor** como un `String`. El argumento se pasa a `getParameter` como un `String` que contiene el nombre del parámetro en la etiqueta `param`. Si no existe la etiqueta `param` que contiene el parámetro especificado, `getParameter` devuelve `null`.
- Un mapa de imágenes es una imagen que no tiene áreas sensibles en las cuales, el usuario puede hacer clic para llevar a cabo una tarea tal como la carga de una página Web diferente en un navegador.

## TERMINOLOGÍA

altura de una imagen	extensión de nombre de archivo	método <code>getImage</code> de la clase
ancho de una imagen	<code>.mid</code>	<code>ImageIcon</code>
animación	extensión de nombre de archivo	método <code>getImageLoadStatus</code>
animación de una serie de imágenes	<code>.rmi</code>	método <code>getParameter</code>
archivo AIFF de Macintosh (. <code>aif</code> o <code>aiff</code> )	extensión de archivo <code>.wav</code>	de la clase <code>Applet</code>
archivo de sonido de Sun (. <code>au</code> )	formato de archivo de sonido	método <code>getWidth</code> de la clase
archivo HTML	de Sun (. <code>au</code> )	<code>Component</code>
archivo Wave de Windows (. <code>wav</code> )	Formato de Intercambio	método <code>loop</code> de la interfaz
área sensible de un mapa de imágenes	de Gráficos (GIF)	<code>AudioClip</code>
atributo de nombre de la etiqueta <code>param</code>	gráficos con doble búfer	<code>MediaTracker.COMPLETE</code>
atributo <code>value</code> de la etiqueta <code>param</code>	Grupo unido de expertos	método <code>paintIcon</code> de la clase
búfer fuera de pantalla	en fotografía (JPEG)	<code>ImageIcon</code>
clase <code>Image</code>	imágenes	método <code>play</code> de la clase
clase <code>ImageIcon</code>	imagen fuera de pantalla	<code>Applet</code>
clase <code>MediaTracker</code>	imagen GIF entrelazada	método <code>play</code> de la interfaz
clase <code>Timer</code>	imagen GIF no entrelazada	<code>AudioClip</code>
clase <code>URL</code>	interfaz <code>ImageObserver</code>	método <code>repaint</code> de la clase
clip de audio	Localizador Uniforme de Recursos	<code>Component</code>
contexto gráfico fuera depantalla	(URL)	método <code>restart</code> de la clase
equilibrio espacio/tiempo	mapa de imágenes	<code>Timer</code>
escalar una imagen	método <code>drawImage</code> de la clase	motor de audio
etiqueta <code>param</code>	<code>Graphics</code>	método <code>start</code> de la clase
extensión de nombre de archivo .aif	método <code>getAudioClip</code> de la	<code>Timer</code>
extensión de nombre de archivo .aiff	clase <code>Applet</code>	método <code>stop</code> de la clase
extensión de nombre de archivo .au	método <code>getCodeBase</code> de la clase	<code>Timer</code>
extensión de nombre de archivo .gif	<code>Applet</code>	método <code>stop</code> de la interfaz
extensión de nombre de archivo .jpeg	método <code>getDocumentBase</code>	<code>AudioClip</code>
extensión de nombre de archivo .jpg	de la clase <code>Applet</code>	método <code>update</code> de la clase
	método <code>getHeight</code> de la clase	<code>Component</code>
	<code>Component</code>	multimedia
	método <code>getIconHeight</code> de la	<code>param</code>
	clase <code>ImageIcon</code>	personalización de un applet
	método <code>getIconWidth</code>	reducción del parpadeo de una
	de la clase <code>ImageIcon</code>	animación
	método <code>getImage</code> de la clase	sistema de ayuda de burbuja
	<code>Applet</code>	sonido

## BUENAS PRÁCTICAS DE PROGRAMACIÓN

- 30.1 Cuando reproduzca sonidos en un applet o en una aplicación, proporcione un mecanismo para que el usuario pueda deshabilitar el sonido.
- 30.2 En un applet, siempre despliegue algo mientras se cargan las imágenes. Entre más tiempo tenga que esperar un usuario para ver información en la pantalla, es más probable que abandone la página Web antes de que la información aparezca.

## OBSERVACIONES DE APARIENCIA VISUAL

- 30.1 Los componentes Swing redefinen el método `update` para evitar que se limpie el fondo (en el caso de componentes transparentes), en respuesta a mensajes `repaint`.
- 30.2 Los componentes GUI de Swing se interpretan utilizando gráficos con doble búfer, de manera predeterminada.

## TIPS DE RENDIMIENTO

- 30.1 Es más eficiente cargar los marcos de la animación como una imagen, que cargar cada imagen por separado (puede utilizar un programa de dibujo para combinar los marcos de la animación dentro de la imagen). Si las imágenes se cargan desde la World Wide Web, cada imagen cargada requiere una conexión separada hacia el sitio en donde se almacenan las imágenes.
- 30.2 Cargar todos los marcos de la animación como una imagen grande podría obligar a su programa a esperar para empezar a desplegar la animación.
- 30.3 Algunas personas que tienen experiencia con objetos `MediaTracker` han reportado que éstos tienen un efecto que va en detrimento del rendimiento. Mantenga esto en mente, como un área que analizará si necesita poner a punto sus aplicaciones multimedia.
- 30.4 Utilizar el método `waitForAll` de `MediaTracker` para esperar a que todas las imágenes registradas se descarguen completamente puede resultar en un gran retraso una vez que el programa comienza la ejecución y hasta que las imágenes en realidad se despliegan. Entre más grandes sean las imágenes, mayor será el tiempo que el usuario tendrá que esperar. Utilice el método `waitForAll` sólo para esperar que un número pequeño de imágenes se desplieguen completamente.
- 30.5 El doble búfer puede reducir o eliminar el parpadeo de una animación, pero puede disminuir visiblemente la velocidad a la que se ejecuta la animación.

## TIP DE PORTABILIDAD

- 30.1 La clase `Image` es una clase `abstract`, por lo que no pueden crearse objetos de `Image` de manera directa. Para lograr la independencia de la plataforma, la implementación de Java en cada plataforma proporciona su propia subclase de `Image` para almacenar la información de la imagen.

## OBSERVACIONES DE INGENIERÍA DE SOFTWARE

- 30.1 Cuando genere una animación para utilizarla en un applet, proporcione un mecanismo para deshabilitarla cuando el usuario navegue una nueva página Web diferente a la página en la que el applet de la animación reside.
- 30.2 La clase `ImageIcon` utiliza un objeto `MediaTracker` para determinar el estado de la imagen que está cargando.

## EJERCICIOS DE AUTOEVALUACIÓN

- 30.1 Complete los espacios en blanco:
  - a) El método \_\_\_\_\_ de `Applet` carga la imagen dentro de un applet.
  - b) El método \_\_\_\_\_ de `Applet` devuelve como un objeto de la clase `URL` a la ubicación en Internet del archivo HTML que invocó al applet.
  - c) Una \_\_\_\_\_ es un formato estándar para una dirección de una pieza de información en Internet.
  - d) El método \_\_\_\_\_ de `Graphics` despliega una imagen de un objeto.
  - e) Con la técnica de \_\_\_\_\_, mientras el programa interpreta una imagen en la pantalla, podría construir la siguiente imagen en un búfer fuera de pantalla. Entonces, cuando es tiempo para desplegar la siguiente imagen, ésta puede colocarse suavemente en la pantalla.
  - f) Conforme se despliega una imagen \_\_\_\_\_, ésta aparece desvanecida mientras el primer lote de filas dibuja un borrador de la imagen y los lotes subsiguientes de filas refinan la imagen desplegada hasta que se completa la imagen.
  - g) Existen dos piezas clave para implementar un gráfico de doble búfer, una referencia a \_\_\_\_\_ y una referencia a \_\_\_\_\_. La primera es donde se desplegarán los píxeles reales; la segunda se utiliza para dibujar los píxeles.

- h) Las imágenes pueden registrarse con un objeto \_\_\_\_\_ para permitir al programa determinar cuando una imagen se cargó por completo.
- i) Java proporciona dos mecanismos para reproducir sonidos en un applet, el método **play** de **Applet** y el método **play** de la interfaz \_\_\_\_\_.
- j) Un \_\_\_\_\_ es una imagen que contiene *áreas sensibles* en las que el usuario puede hacer clic para llevar a cabo una tarea, tal como la carga de una página Web diferente.
- k) El método \_\_\_\_\_ de la clase **ImageIcon** despliega la imagen de **ImageIcon**.

**30.2**

Establezca si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

- a) En la actualidad, Java soporta dos formatos de imagen. Los nombres de archivos de estos tipos terminan con **.jfif** o **.gpg** respectivamente.
- b) Redefinir el método **update** del applet para llamar a **paint** sin limpiar el applet, reducirá significativamente el parpadeo de la animación.
- c) Un sonido será depositado en la basura tan pronto como termine la reproducción.
- d) Los componentes GUI de Swing contienen gráficos internos con doble búfer.

## RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

- 30.1** a) **getImage**. b) **getDocumentBase**. c) **URL**. d) **drawImage**. e) Gráficos con doble búfer. f) Entrelazada. g) **Image**, **Graphics**. h) **MediaTracker**. i) **AudioClip**. j) Mapa de imágenes. k) **paintIcon**.
- 30.2** a) Falso, debe ser **.gif** o **.jpg**. b) Verdadero. c) Falso, el sonido se marcará para el recolector de basura (si no está referenciado por un **AudioClip**) y se arrojará a la basura cuando el recolector de basura sea capaz de ejecutarse. d) Verdadero.

## EJERCICIOS

- 30.3** Describa cómo hacer una animación “amigable para el navegador”.
- 30.4** Explique los distintos aspectos de la eliminación del parpadeo en Java.
- 30.5** Explique la técnica de los gráficos con doble búfer.
- 30.6** Describa los métodos de Java para reproducir y manipular los clips de audio.
- 30.7** (*Animación.*) Elabore un programa de animación en Java de propósito general. Su programa debe permitir al usuario especificar la secuencia de marcos a desplegar, la velocidad a la cual se despliegan las imágenes, los sonidos a reproducir mientras se ejecuta la aplicación, etcétera.
- 30.8** (*Protector de pantalla.*) Utilice la animación de una serie de sus imágenes favoritas para crear un programa protector de pantalla. Elabore distintos efectos especiales que aprovechen la imagen, que la hagan girar, que la desvanezcan, que muevan la imagen hacia los límites de la pantalla y otras cosas similares.
- 30.9** (*Borrar una imagen al azar.*) Suponga que se despliega una imagen en un área rectangular de la pantalla. Una manera de eliminar la imagen es establecer inmediatamente cada píxel con el mismo color, pero esto tiene un efecto visual monótono. Escriba un programa en Java que despliegue una imagen y que la elimine mediante la generación de números aleatorios para seleccionar los pixeles individuales a eliminar. Una vez que se eliminó la mayor parte de la imagen, elimine todos los pixeles restantes al mismo tiempo. Usted puede hacer referencia a los pixeles individuales haciendo que una línea comience y termine en el mismo punto. Puede intentar distintas variantes de este problema. Por ejemplo, podría desplegar las líneas de manera aleatoria, o podría desplegar las figuras al azar para eliminar regiones de la pantalla.
- 30.10** (*Texto intermitente.*) Elabore un programa en Java que repita intermitentemente texto en la pantalla. Haga esto entremezclando un texto con una imagen plana de color como fondo. Permita al usuario controlar la “velocidad de parpadeo” y el color de fondo o patrón.
- 30.11** (*Instantánea de imágenes.*) Elabore un programa en Java que coloque una instantánea de una imagen en la pantalla. Haga esto mediante la mezcla de una imagen con una imagen plana de color como fondo.
- 30.12** (*Reloj digital.*) Implemente un programa que despliegue un reloj digital en la pantalla. Podría agregar opciones para escalar el reloj; desplegar el día, el mes y el año; emitir un sonido de alarma; reproducir ciertos sonidos en horas predefinidas y cosas similares.
- 30.13** (*Llamar la atención hacia una imagen.*) Si usted desea enfatizar una imagen, puede colocar una fila simulada de bulbos de luz alrededor de la imagen. Puede dejar los bulbos encender y apagar al azar, o puede dejarlos encender y apagar uno después del otro.
- 30.14** (*Zoom de imagen.*) Elabore un programa que le permita hacer acercamientos, o alejamientos de una imagen.