¡Domine la sintaxis completa Java 2.0!

Utilice etiquetas, botones, selectores, listas, barras de progreso, divisores, separadores, cuadros de diálogo, tablas y mucho más.

Conquiste la interfaz de usuario Swing de Java.



JAVA 2

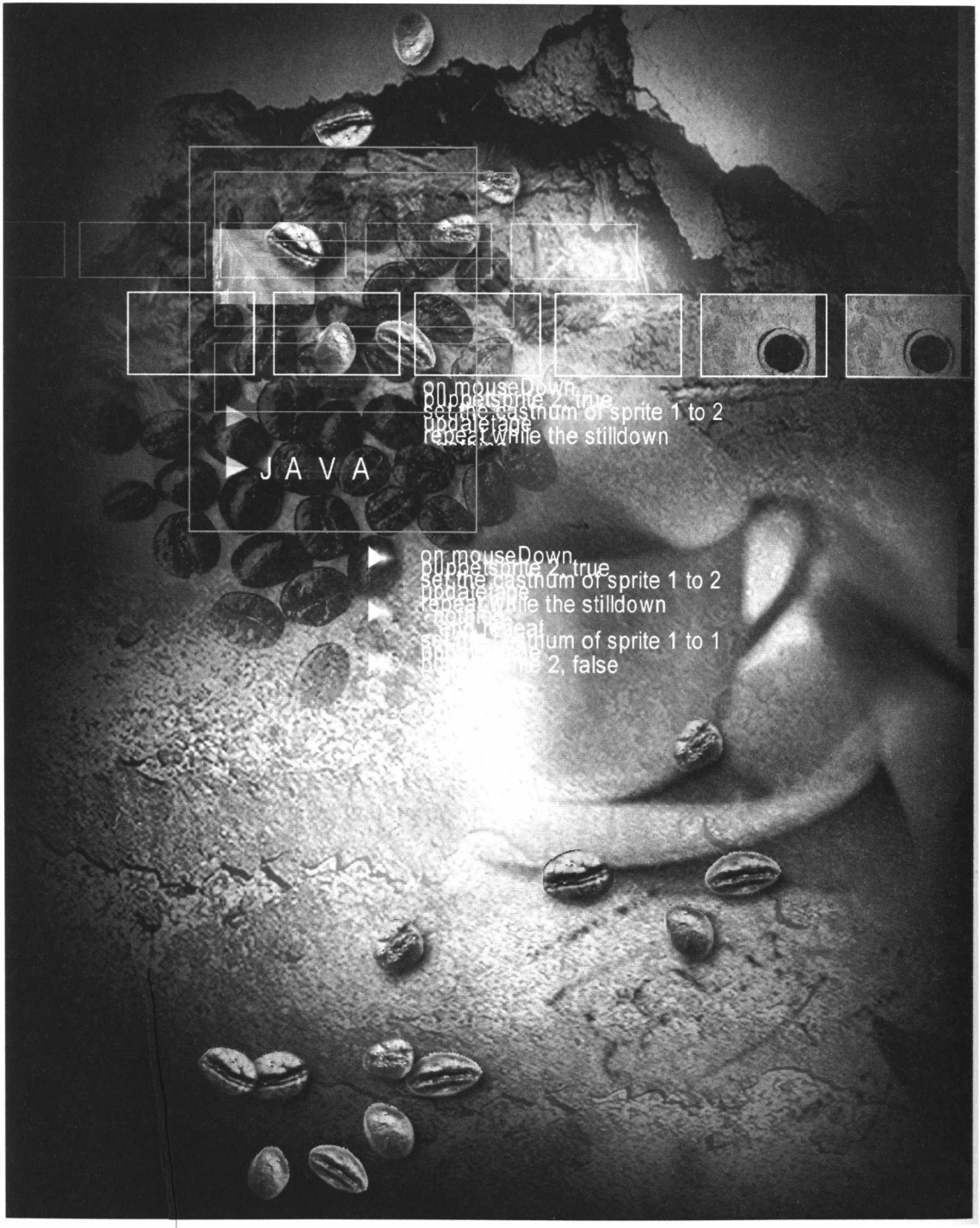
CD-ROM

con el código, imágenes y otros ficheros utilizados en el libro, así como un gran número de utilidades.

Steven Holzner

ANAYA MULTIMEDIA

CORIOLIS


JAVA
on mouseDown
set the castNum of sprite 1 to 2
updateStage
repeat while the stillDown
on mouseDown
set the castNum of sprite 1 to 2
updateStage
repeat while the stillDown

Índice

Introducción **23**

Contenido del libro 24
Requerimientos 25
Otros recursos 26

1. Java básico **29**

Todo sobre Java 30
Orígenes del lenguaje Java 32
Todo sobre bytecodes 32
La seguridad del lenguaje Java 33
Programas Java 34
¿Es Java 2 o Java 1.2? 37
Adquirir e instalar Java 37
¿Qué ocurre con CLASSPATH? 38
¿Cuáles son las novedades de Java 1.1? 39
¿Qué está censurado en Java 1.1? 41
¿Cuáles son las novedades de Java 2? 41
¿Qué se censuró en Java 2? 44
Escribir código: creación de archivos de código 44

Escribir código: conocer las palabras reservadas de Java 45
Escribir código: crear una aplicación 48
public class app 49
public static void main[String[] args) 50
System.out.println("¡Hola desde Java!"); 51
Compilación 51
Compilación: utilizando opciones en la línea de comandos 52
Opciones de compilación cruzada 55
Compilación: revisión de los métodos censurados 55
Ejecución del código 56
Ejecución de código: utilizar las opciones de la línea de comandos 59
Conocimientos básicos: comentar el código 61
Conocimientos básicos: importando paquetes y clases Java 64
Conocimientos básicos: buscar clases Java con CLASSPATH 66
Crear applets 69
Ejecutar applets 71
Crear aplicaciones ventana 72
Ejecutar aplicaciones ventana 73
Diseño de programas Java 74
Rendimiento 75
Mantenimiento 75
Extensibilidad 76
Disponibilidad 76
Distribución del programa Java 77

2. Variables, arrays y cadenas 79

Variables 79
Tipos de datos 81
Arrays 82
Cadenas 85
¿De qué tipo de datos disponemos? 87
Creación de literales enteros 88
Creación de literales en coma flotante 89
Creación de literales booleanos 91
Creación de literales carácter 91
Creación de literales tipo cadena 93
Declaración de variables de tipo entero 93

Declaración de variables de tipo coma flotante 94
Declaración de variables de tipo carácter 95
Declaración de variables de tipo booleano 96
Inicialización de variables 98
Inicialización dinámica 99
Conversión de tipos de datos 100
Conversiones automáticas 100
Casting a nuevos tipos de datos 101
Declaración de arrays unidimensionales 103
Creación de arrays unidimensionales 104
Inicialización de arrays unidimensionales 105
Declaración de arrays multidimensionales 105
Creación de arrays multidimensionales 106
Inicialización de arrays multidimensionales 108
Creación de arrays multidimensionales 109
Longitud de un array 110
La clase String 110
Creación de cadenas 117
Obtención de la longitud de la cadena 119
Concatenación de cadenas 120
Obtención de caracteres y substring 121
Búsqueda y reemplazamientos en cadenas 122
Cambio de mayúsculas a minúsculas (o viceversa) en cadenas 123
Formateo de números en cadenas 124
La clase StringBufferr 124
Creación de StringBuffers 125
Obtención y establecimiento de longitudes y capacidades de StringBuffer .. 129
Establecer caracteres en StringBuffers 130
Añadir e insertar utilizando StringBuffers 130
Borrar texto en StringBuffers 131
Reemplazar texto en StringBuffers 132

3. Operadores, condicionales y bucles 135

Operadores 135
Condicionales 137
Bucles 139
Precedencia de operadores 140

Incremento y decremento: ++ y -- 141
NOT unario: ~ y ! 143
Multiplicación y división: * y / 144
Módulo: % 145
Suma y resta: + y - 145
Operadores de desplazamiento: >>, >>> y << 146
Operadores de relación: >, >=, <, <=, == y != 147
Operadores lógicos a nivel de bit AND, Xor y OR: &, ^ y | 148
&& y || lógicos 151
El operador If-Then-Else: ?: 153
Operadores de asignación: = y [operador]= 154
Utilización de la clase Math 156
Comparación de cadenas 158
La sentencia if 159
La sentencia else 160
If anidados 161
Escalas if-else 161
La sentencia switch 162
Bucle while 164
Bucle do-while 167
Bucle for 168
Bucles anidados 171
Sentencia break 172
Sentencia continue 174

4. Programación orientada a objetos 177

Clases 179
Objetos 179
Miembros de datos 180
Métodos 181
Herencia 182
Declaración y creación de objetos 182
Declarar y definir clases 186
Crear variables de instancia 188
Acceso a variables 190
Crear variables de clase 191
Crear métodos 193

Establecer el acceso a los métodos 195
Pasar parámetros a los métodos 196
Argumentos de la línea de comandos pasados a main 198
Devolver valores desde los métodos 199
Crear métodos de clase 201
Crear métodos de acceso a datos 202
Crear constructores 204
Pasar parámetros a constructores 205
Un ejemplo completo de clase 205
Comprender el alcance de las variables 207
Recursividad 209
Colección garbage y gestión de memoria 210
Evitar las referencias circulares 212
Colección garbage y el método finalize 213
Sobrecarga de métodos 214
Sobrecarga de constructores 215
Pasar objetos a métodos 217
Pasar arrays a métodos 219
Usar la palabra clave this 220
Devolver objetos desde métodos 221
Devolver arrays desde métodos 222

5. Herencia, clases internas e interfaces 225

¿Por qué la herencia? 226
¿Por qué las interfaces? 227
¿Por qué las clases internas? 228
Crear una subclase 229
Especificadores de acceso y herencia 231
Llamar a los constructores de la superclase 234
Crear multiniveles de herencia 237
Gestionar multiniveles de constructores 239
Sobrescritura de métodos 241
Acceso a los miembros sobrescritos 242
Usar variables de superclase con objetos de subclases 244
Dispatching dinámico de métodos (Polimorfismo en tiempo de ejecución) . 246
Crear clases abstractas 249

Abandonar la sobrescritura con final 251
Abandonar la herencia con final 252
Crear constantes con final 253
Relación es-a frente a tiene-a 253
La clase Object de Java 255
Usar interfaces para herencia múltiple 258
Crear clases internas 261
Crear clases internas anónimas 262

6. AWT: Applets, aplicaciones y gestión de eventos 265

Abstract Windowing Toolkit 266
Applets 267
Aplicaciones 269
Usar Abstract Windowing Toolkit 270
Crear Applets 284
Usar la etiqueta HTML <APPLET> 287
Gestionar browsers no Java 290
Introducir etiquetas <APPLET> en el código 290
Usar los métodos init, start, stop, destroy, paint y update 291
Dibujar gráficos en applets 293
Usar el plug-in de Java del browser 294
Leer parámetros en applets 295
Usar las consolas de Java en los browsers 297
Añadir controles a las applets: Campos de texto 297
Añadir controles a las applets: botones 300
Gestión de eventos 300
  Gestión de eventos estándar 302
  Uso de clases delegadas 304
  Uso de los comandos de acción 306
  La forma antigua de gestionar eventos 307
  Extender componentes 307
Usar las clases adaptador 309
Usar clases adaptador internas anónimas 311
Crear ventanas de aplicación 312
Salir de una aplicación al cerrar su ventana 316
Aplicaciones que se pueden ejecutar como applets 317

7. AWT: Cuadros de texto, botones, casillas de activación y plantillas 321

Cuadros de texto 321
Botones 322
Casillas de activación 322
Botones de opción 322
Plantillas 323
Usar cuadros de texto 324
Usar etiquetas 327
Usar botones 329
Usar casillas de activación 334
Usar botones de opción 339
Esquemas de flujo (flow layout) 341
Grid layouts 345
Usar paneles 349
Border Layout 351
Card Layouts 355
Grid bag layouts 358
Usar intercalados y rellenos 365
Crear el propio gestor de esquemas 367

8. AWT: Listas, cuadros de lista, áreas de texto, barras y cuadros de desplazamiento 371

Listas 371
Cuadros de lista desplegables 372
Áreas de texto 372
Barras de desplazamiento 373
Paneles de desplazamiento 373
Usar las áreas de texto 374
Reemplazar texto en áreas de texto 377
Buscar y seleccionar texto en áreas de texto 379
Usar listas 381
Usar listas de selección múltiple 388
Usar cuadros de lista desplegables 391
Usar barras de desplazamiento 396

Barras de desplazamiento y border layouts 404
Usar cuadros de desplazamiento 407

9. AWT: Gráficos, imágenes, texto y fuentes 415

Gráficos 415
Imágenes 415
Texto y fuentes 416
Teclado y ratón 416
Usar el ratón 416
Usar el teclado 420
Usar fuentes 425
Usar imágenes 434
Redimensionar imágenes 437
Dibujar gráficos 439
Dibujar rectas 447
Dibujar óvalos 447
Dibujar rectángulos 447
Dibujar rectángulos redondeados 448
Dibujo libre 449
Dibujar arcos 450
Dibujar polígonos 450
Establecer los modos de dibujo 450
Seleccionar colores 450
Usar Canvases 454
Usar la interfaz ImageObserver 456
Usar la clase MediaTracker 458
Trabajar pixel por pixel: Las clases PixelGrabber y MemoryImageSource .. 462
Dar brillo a las imágenes 466
Convertir imágenes a escala de grises 467
Realzar imágenes 469

10. AWT: Ventanas, menús y cuadros de diálogo 473

Ventanas 473
Menús 474
Cuadros de diálogo 475
Crear ventanas Frame 475
Mostrar y ocultar ventanas 477

Gestionar eventos de ventana 480
Ocultar ventanas automáticamente al cerrarlas 483
Usar la clase Window 484
Crear menús 489
Crear un objeto MenuBar 491
Crear objetos Menu 493
Crear objetos MenuItem 495
Gestionar los eventos de menú 498
Más opciones de menú 500
Añadir separadores de menú 503
Deshabilitar elementos de menú 503
Añadir marcas de activación a menús 505
Crear submenús 508
Menús emergentes 510
Cuadros de diálogo 512
Cuadros de diálogo de archivos 518

11. Swing: Applets, aplicaciones y cambios de apariencia 523

Clases Foundation de Java 523
Swing 524
Componentes peso pesado contra peso ligero 527
Características Swing 528
Utilizar paneles en la programación de gráficos 529
Arquitectura Modelo-Vista-Controlador 530
Trabajar con Swing 531
Preparar para crear un applet Swing 536
Comprender los root panes 539
Comprender layered panes 542
Comprender los content panes 545
Crear un applet Swing 545
Pintar en Swing frente a AWT 545
Visualizar controles en Swing frente a AWT 546
Usar la clase JPanel 546
Crear una aplicación Swing 549
Cerrar ventanas JFrame 553
Seleccionar los bordes del componente 555

Usar Insets 555
Establecer la apariencia 560
Establecer los componentes para la apariencia 565

12. Swing: Cuadros de texto, botones y casillas de activación 571

Etiquetas y cuadros de texto 571
Botones 572
Botones toggle 572
Casillas de activación y botones de opción 572
Usar etiquetas 573
Usar iconos imagen 577
Usar imágenes en etiquetas 579
Usar cuadros de texto 580
Abstract Button: base de los botones Swing 583
Usar botones 589
Visualizar imágenes en botones 592
Usar imágenes rollover y deshabilitadas 594
Botones por defecto y mnemónicos 595
Usar botones toggle 597
Crear grupos de botones toggle 600
Usar casillas de activación 601
Usar botones de opción 604
Usar imágenes en casillas de activación y botones de opción 607
Obtener y fijar el estado de las casillas de activación y de los botones de opción 608

13. Swing: viewports, desplazamiento, deslizadores y listas 613

Viewports 613
Paneles de desplazamiento 614
Deslizadores 614
Barras de desplazamiento 614
Listas 614
Manejo de viewports 615
Creación de paneles de desplazamiento 621
Creación de paneles de desplazamiento con cabeceras y bordes 627
Desplazamiento de imágenes 629
Creación de deslizadores 630

Relleno de un deslizador 636
Pintar las marcas de un deslizador 637
Pintar etiquetas en un deslizador 638
Ajuste de la extensión del deslizador 639
Creación de barras de desplazamiento 640
Creación de listas 646
Gestión de selecciones múltiples 653
Modos de selección de lista 653
Visualización de imágenes en listas 655
Creación de un modelo de lista personalizado 657
Creación de un renderizador personalizado para celdas de lista 657
Procesamiento de doble clic en listas 658

14. Swing: barras, herramientas, cuadros, separadores y selectores 663

Cuadros combinados 663
Barras de progreso 664
Selectores 664
Herramientas de ayuda 665
Separadores 665
Creación de cuadros combinados 665
Manejo de los eventos de selección del cuadro combinado 672
Creación de cuadros combinados editables 674
Adición de imágenes a cuadros combinados 676
Creación de un modelo de cuadro combinado 678
Creación de un renderizador personalizado para el cuadro combinado 678
Creación de barras de progreso 679
Actualización de barras de progreso 684
Manejo de los eventos de barras de progreso 686
Creación de ayudas de herramientas 687
Creación de separadores 690
Cambio de tamaño automático de separadores 693
Creación de un selector de color 695
Creación de selectores de archivos 699
Creación de filtros para selectores de archivo 709

15. Swing: paneles de capas, de lengüetas, separadores y distribuciones 715

Paneles de capas 716
Paneles de lengüetas 716
Paneles de separación 716
Distribución 717
Comprensión de los componentes de la Swing y el orden Z 717
Transparencia en los componentes de la Swing 719
Uso de paneles de capas 722
Creación de paneles de lengüetas 725
Especificación de la posición de las lengüetas en los paneles de lengüetas 732
Uso de paneles de separación 734
Paneles de separación expandibles con un clic 740
Configuración de la orientación del panel de separación 742
Configuración del tamaño del divisor de un panel de separación 744
Uso del gestor de distribución de cuadro 745
Uso de la clase Box 748
Uso del gestor de distribución de superposición 752

16. Swing: menús y barras de herramientas 757

Menús 757
Barras de herramientas 758
Crear una barra de menús 758
Crear un menú 761
Crear un elemento de menú 765
Crear un sistema de menús básico 768
Adición de imágenes a menús 772
Crear elementos de menú de casillas de verificación 774
Crear menús de botones de activación 777
Crear submenús 780
Crear aceleradores y mnemónicos de menú 782
Habilitar/inhabilitar elementos de menú y cambiar títulos en tiempo de ejecución 785
Añadir y eliminar elementos de menú en tiempo de ejecución 788
Añadir botones y otros controles a menús 790

Crear menús emergentes 791
Crear barras de herramientas 797
Añadir cuadros combinados y otros controles a barras de herramientas 801

17. Swing: ventanas, paneles, marcos internos y cuadros de diálogo 805

Ventanas 805
Cuadros de diálogo 806
Crear una ventana 806
Crear una ventana de marco 810
Crear un panel de escritorio 812
Crear marcos internos 814
Uso de JOptionPane para crear cuadros de diálogo 825
Crear cuadros de diálogo con panel de opciones de confirmación 834
Crear cuadros de diálogo con panel de opciones de mensaje 835
Crear cuadros de diálogo con panel de opciones de campo de texto de entrada 837
Crear cuadros de diálogo con panel de opciones para entrada de cuadros combinados 839
Crear cuadros de diálogo con panel de opciones de marcos internos 841
Crear cuadros de diálogo con JDialog 843
Obtener entrada de los cuadros de diálogo creados con JDialog 848

18. Swing: tablas y árboles 851

Tablas 851
Árboles 852
Crear tablas 852
Crear árboles 871

19. Swing: Componentes de texto 885

Crear componentes de texto en Swing: la clase JTextComponent 885
Crear campos de texto 885
Ajustar la alineación del campo de texto 886
Desplazar campos de texto 889
Crear campos de palabra clave 890

20. Stream I/O y archivos 895

Usar la clase File 895
Trabajar con InputStream 897
Trabajar con OutputStream 897
Trabajar con FileInputStream 897
Trabajar con FileOutputStream 899
Trabajar con ByteArrayInputStream 900
Trabajar con ByteArrayOutputStream 901

21. Programación multihilo y animación 905

Usar hilos en Java 905
Obtener el hilo principal 907
Dar nombre a un hilo 907
Detener un hilo 908
Crear un hilo con la interfaz Runnable 909
Crear un hilo con la clase Thread 911
Crear hilos múltiples 916
Espera (para unión) de hilos 917
Comprobar que un hilo está vivo 919
Asignar la prioridad y detención de hilos 920
¿Por qué utilizar la sincronización? 923
Sincronizar bloques de código 924
Métodos sincronizados 926
Comunicación entre hilos 927
Suspender y reanudar hilos 929
Crear gráficos animados con hilos 931
Eliminar el parpadeo en animaciones gráficas 934
Suspender y reanudar animaciones gráficas 935
Doble buffer 937

22. Creación de paquetes, interfaces, archivos JAR y Java Beans 941

Crear un paquete 941
Crear paquetes que contienen paquetes 943
Crear una interfaz 944
Implementación parcial de una interfaz 945

Crear un archivo JAR 946
Obtener los contenidos del archivo JAR 948
Extraer archivos desde un archivo JAR 948

Apéndice. Contenido del CD-ROM 951

Índice alfabético 953

Introducción

Bienvenido a la biblia de Java 2. Este libro se ha diseñado para que sea todo lo comprensible y accesible que es posible teniendo en cuenta que es un libro de Java. En él va a encontrar tanto Java como quepa en sus páginas.

Java no es un lenguaje de programación ordinario: inspira devoción, pasión, exaltación y excentricidad (no se menciona exasperación ni frustración). Esperemos que lo que Java tiene que ofrecer le resulte irresistible al igual que ha ocurrido con otros muchos programadores (de hecho, la programación en Java es uno de los conocimientos más lucrativos que se pueden tener hoy en día).

Al lenguaje de programación Java se le ha llamado "C++ para Internet", y aunque hay algo de verdad en eso, Internet no es el único lugar en el que actualmente se encuentra Java. Cada vez hay más empresas que utilizan el lenguaje de programación Java para construir aplicaciones que no tienen relación con Internet, pero que tienen que ser independientes de la plataforma. He visto que muchas de las grandes corporaciones han cambiado gradualmente su programación interna de C++ a Java. La influencia del lenguaje de programación Java se está extendiendo y no hay visos de pararlo, y con cada versión se tiene más poder y más profundidad para trabajar con este lenguaje.

Si está de acuerdo conmigo, se sentirá atraído por la programación en Java, ya que lo que puede hacer con este lenguaje es asombroso. Verá lo que quiero decir en cada una de las páginas de este libro.

Contenido del libro

Este libro está diseñado para mostrarle toda la historia del lenguaje de programación Java que un libro puede contener. Veremos no sólo toda la sintaxis de Java, desde la declaración de variables hasta temas de orientación a objetos avanzada, sino también el lenguaje Java en el mundo real. Se cubrirán temas como permisos de acceso a *applets*, uso del visualizador de Java, creación de conexiones cliente/servidor sobre Internet, creación de *Java Beans*, conexión de bases de datos y *multithread.*

En este libro se tratan cientos de temas, y cada uno de ellos viene con un ejemplo que muestra cómo funciona. Está dividido en temas separados y fácilmente accesibles y en cada uno de ellos se trata un caso de programación diferente. Algunos de estos temas son los que se muestran a continuación:

- toda la sintaxis de Java 2
- programación orientada a objetos
- herencia y clases internas
- *abstract windowing toolkit* (AWT)
- botones, casillas de activación y botones de opción
- selectores, listas y cuadros de lista desplegables
- gráficos, imágenes, texto y fuentes
- menús, cuadros de diálogo y ventanas
- barras de progreso, barras de desplazamiento, separadores y cuadros de desplazamiento
- tratamiento y ajuste de imágenes
- *Java Swing*
- cambios de apariencia con *Swing*
- todos los componentes *Swing*
- componentes de texto *Swing*
- colecciones Java

- *multithreads*
- flujos de E/S
- manejo de ficheros
- redes y *sockets*
- árboles y tablas
- *Java Beans*
- paquetes, interfaces y ficheros JAR
- seguridad
- manejo de excepciones
- manejo del teclado y del ratón

Esta lista no es completa. Más adelante trataremos muchos temas más. Uno al que prestaremos especial atención y que no se trata en la mayoría de los libros, es *Java Swing*, la nueva y revolucionaria interfaz para la programación de clases Java.

Existe una norma que se usará en este libro y que deberá conocer: para resaltar una línea concreta de código, se sombreará de la siguiente forma:

```
public class app
{
  public static void main(String[] args)
  {
      (new printer( )).print( );
  }
}
```

Requerimientos

En este libro, utilizará Java 2, versión 1.2.2. Si no usa esta versión como mínimo, puede que al ejecutar el código aparezcan algunos errores. Por ello, obtenga *Java Development Kit* (JDK) en http://java.sun.com/products/jdk/1.2 (o la última versión).

También necesitará tener una forma de crear programas Java. Estos programas son ficheros planos de texto con instrucciones y declaraciones Java. Para crear un programa Java, deberá tener un editor que permita guardar ficheros en formato texto. Para más detalles, vea el capítulo 1.

Todo lo que necesita para usar este libro, además del editor de programas, lo puede obtener en el sitio Web de Sun Java, http://java.sun.com. JDK tiene

todo lo que necesita para crear *applets* y aplicaciones estándares de Java e incluso posee un visualizador de *applets* para verlos mientras está trabajando.

Además de JDK, usaré *Beans Development Kit* (BDK), y *Java Servlet Development Kit* (JSDK), que también se pueden obtener en el sitio Web de Java. Si quiere continuar con la programación de base de datos incluida en el libro, necesitará crear en su máquina una fuente de datos ODBC. En el CD encontrará la base de datos que trabaja con esta fuente de datos, junto con el código, imágenes y otros ficheros utilizados en el libro, sin mencionar un gran número de utilidades.

Finalmente, Java le ofrece gran cantidad de documentación, cientos de libros de calidad. Esa documentación está almacenada en páginas HTML enlazadas y para trabajar con ellas y visualizarlas, necesitará tener un navegador Web.

Otros recursos

Hay otros recursos Java que pueden servir de ayuda con el lenguaje de programación Java. Como se mencionó anteriormente, hay decenas de miles de páginas de documentación que vienen con el mismo lenguaje Java. Hay además muchas, muchas páginas Web referentes al lenguaje Java (haciendo una búsqueda aleatoria en la Web se encuentran unas 10.268.200 páginas que mencionan Java; de hecho, buscando "*Java tutorial*" se encuentran unas 11.614 páginas). A continuación se citan algunos de estos recursos útiles:

- página principal de Java en http://java.sun.com
- guía didáctica de Sun Java en http://java.sun.com/docs/books/tutorial
- soporte técnico en línea de Sun en http://java.sun.com/products/jdk/1.2/docs/index.html
- para obtener el lenguaje de programación Java, ir a http://java.sun.com/products/jdk/1.2

Entre otros temas, puede encontrar guías didácticas sobre los conceptos siguientes en la página de tutorías de Sun http://java.sun.com/dosc/books/tutorial:

- colecciones
- internacionalización
- *servlets*

- gráficos 2D
- seguridad en JDK 1.2
- sonido
- ficheros *Java Archive* (JAR)
- *Java Beans*
- conectividad de bases de datos Java (JDBC)
- interfaz nativa de Java
- llamada remota a métodos (RMI)
- reflexión

Además hay un número de grupos *Usenet* para programadores de Java, que incluyen:

- comp.lang.java.advocacy
- comp.lang.java.announce
- comp.lang.java.beans
- comp.lang.java.corba
- comp.lang.java.databases
- comp.lang.java.gui
- comp.lang.java.help
- comp.lang.java.machine
- comp.lang.java.programmer
- comp.lang.java.security
- comp.lang.java.softwaretools

Para Java existe mucha ayuda, pero espero que no tenga que acudir a ella. Este libro está diseñado para proporcionarle todo lo que necesite (y espero que así sea. Ahora es el momento de empezar a trabajar con Java, comenzando por el capítulo 1.

1 Java básico

Bienvenido a nuestro gran libro de programación Java. En este libro, trararemos con detenimiento y detalle toda aquella programación Java que puede estar contenida en un libro.

Como el propósito de este libro es poner a su disposición todo el lenguaje de programación Java, preparándole para usarlo, no dejaremos de lado los temas difíciles. Habrá algunos paquetes de programación con los que disfrutará más que con otros, y espero que elija Java como plataforma de programación.

Este primer capítulo cubre los conocimientos fundamentales de Java que habrá que recordar para el resto de los capítulos. En los siguientes capítulos, verá la ejecución de código Java, pero ninguno es útil si no puede ejecutar y crear programas Java. Ese conjunto básico de conocimientos, la creación y ejecución de programas Java, es el tema de este capítulo, y en los siguientes podrá poner en práctica dicha información para ensayar la sintaxis Java que desarrollaremos.

En este capítulo, vamos a trabajar con la mecánica de crear programas Java, desde la instalación hasta la escritura de código, desde asegurarse de que el programa es capaz de encontrar lo que necesita hasta visualizar una simple salida. Estos conocimientos son los que podrá usar en los siguientes capítulos. Estos se centran en todas las interioridades de la escritura de

código Java; este capítulo trata el resto del proceso que hace que el código funcione.

Ya debería conocer mucho sobre el contenido de este capítulo, en cualquier caso se revisará con detenimiento (alguno de ellos está limitado por ser nuevo y a pesar de todo, muy poca gente sabe lo que la línea de comandos del compilador Java puede hacer). Si ya tiene una instalación de Java que funciona y puede escribir y ejecutar un programa básico en Java, ya está familiarizado con la mayor parte de lo que verá en este capítulo. Por consiguiente, puede pasar rápidamente por las páginas siguientes y continuar con el capítulo 2, donde empezaremos a profundizar en la sintaxis de Java, lo que realmente hace que Java funcione. De lo contrario, trabaje con el contenido de este capítulo, pues proporciona la base para varios de los capítulos siguientes.

Todo sobre Java

¿De dónde procede el lenguaje de programación Java y por qué es tan popular? Como otros lenguajes de programación, Java satisfizo una necesidad específica de su tiempo. Antes de que Java apareciera, por ejemplo, C era un lenguaje extremadamente popular entre los programadores y parecía que era el lenguaje de programación perfecto, combinando los mejores elementos de los lenguajes de bajo y alto nivel en un lenguaje de programación que se ajustaba a la arquitectura del ordenador y que gustaba a los programadores.

Sin embargo, el lenguaje C tenía limitaciones, al igual que los lenguajes de programación anteriores. Cuando los programas crecían, los programas C se hacían inmanejables porque no había una forma fácil de acortarlo. Esto quiere decir que el código de la primera línea de un programa largo podría interferir con el código de la última línea, y el programador tendría que recordar todo el código mientras programaba.

La programación orientada a objetos se hizo popular por ser capaz de dividir programas largos en unidades semi-autónomas. El lema de la programación orientada a objetos es "divide y vencerás". En otras palabras, un programa se puede dividir en partes fácilmente identificables. Por ejemplo, supongamos que para mantener fresca la comida utilizara un sistema complejo. Debería comprobar la temperatura de la comida usando un termómetro, y cuando la temperatura fuera lo suficientemente alta, activaría un interruptor que arrancara el compresor e hiciera funcionar las válvulas para que el frío circulara; luego arrancaría un ventilador que moviera el aire. Esa es una forma de hacerlo. Sin embargo, otra consiste en coordinar todas esas operaciones de forma que sean automáticas, cubriendo todo con una unidad senci-

lla, un refrigerador. Ahora las interioridades no se ven y lo único que hay que hacer es introducir o sacar comida del frigorífico.

Así funcionan los objetos: ocultan los detalles de la programación al resto del programa, reduciendo todas las interdependencias que aparecen en un programa C e inicializando una interfaz bien definida y controlable que mantiene la conexión entre el objeto y el resto del código. Así usted puede pensar en el objeto como algo fácil, por ejemplo, puede tener un objeto que gestiona toda la interacción con la pantalla, un objeto al que le llame pantalla. Puede utilizarlo de distintas formas, que verá a lo largo de este libro, para manipular aquello en lo que pretende actuar (en este caso, la pantalla). Después de crear ese objeto, sabe que la pantalla es gestionada por ese objeto y puede tenerlo en mente, pero en ninguna otra parte del código tendrá que gestionar el manejo de la pantalla, podrá usar en su lugar el objeto pantalla que ha creado.

Cuando se añadió al lenguaje C la programación orientada a objetos, nació C++, y los programadores tuvieron un nuevo aliciente. C++ permite a los programadores tratar grandes programas, y el código orientado a objetos ayudó a resolver también muchos de los otros problemas. Por ejemplo, el hecho de soportar objetos lo hacía más fácil para los fabricantes que suministraban software con mucho código reutilizable, preparado para su uso. Para crear un objeto, utiliza una clase que actúa como una plantilla o creador de patrones para ese objeto. Se puede pensar en una clase como un tipo de objeto, al igual que una variable puede ser de tipo entero.

Como C++ soporta clases, los proveedores de software pueden proporcionarle enormes librerías de clases, a partir de las cuales se puede empezar a crear objetos. Por ejemplo, una de las librerías de clases de C++ más populares es la librería Microsoft Foundation Class (MFC) que viene con Visual C++ de Microsoft, y en ella los programadores encontraron una mejora tremenda respecto a los tiempos pasados. Cuando se escribía un programa Windows en C, se necesitaban aproximadamente cinco páginas de código para visualizar una ventana vacía. Sin embargo, utilizando una clase de la librería MFC, simplemente había que crear un objeto del tipo de ventana que se quisiera usar: con o sin borde, como un cuadro de diálogo, etc. El objeto ya tenía toda la funcionalidad del tipo de ventana que quería crear, por lo que para crear una ventana sólo era necesaria una línea de código, la línea en la que se creaba el nuevo objeto ventana de la clase que se había seleccionado.

Más impresionante todavía era el hecho de que se podía usar una clase MFC como clase base para las clases propias, añadiendo la funcionalidad que se quisiera por medio de la herencia de la programación orientada a objetos. Por ejemplo, supongamos que quiere que su ventana tenga una barra de

menús; entonces puede derivar su propia clase de una ventana simple de MFC, añadiendo a esa clase una barra de menús para crear la clase nueva. De esta forma, añadiendo unas cuantas líneas de código a las de los programadores de Microsoft, puede construir su propia clase. (En este libro verá en detalle cómo funciona la programación orientada a objetos).

Todo esto parecía fabuloso a los programadores, y C++ llegó muy lejos. Parecía que el lenguaje de programación perfecto había llegado. ¿Qué podía haber mejor? Sin embargo, el mismo entorno de programación iba a verse sometido a un gran cambio con la popularización de lo que equivale a un nuevo e inmenso entorno de programación: la red Internet. Y eso es lo que hizo que Java fuera tan popular.

Orígenes del lenguaje Java

Originalmente, Java no fue creado para la red Internet. La primera versión de Java empezó en 1991 y fue escrita en 18 meses en Sun Microsystems. De hecho, en ese momento, ni siquiera se llamó Java; se llamó Oak y se utilizó en Sun para uso interno.

La idea original para Oak era crear un lenguaje orientado a objetos independiente de la plataforma. Por entonces, muchos programadores se limitaban a la programación del IBM PC, pero el entorno corporativo podía incluir toda clase de plataformas de programación, desde el PC hasta los grandes sistemas. Lo que había detrás de Oak era crear algo que se pudiera usar en todos los ordenadores (y ahora que Java se ha hecho popular gracias a la red Internet, cada vez más corporaciones están adoptándolo para uso interno en lugar de C++, precisamente por esa razón). El lanzamiento original de Oak no fue especialmente fascinante; Sun quería crear un lenguaje que se pudiera usar en electrónica.

Oak pasó a llamarse Java en 1995, cuando se lanzó para el uso público y supuso un éxito casi inmediato. En ese momento, Java había adoptado un modelo que lo hizo perfecto para la red Internet, el modelo *bytecode*.

Todo sobre *bytecodes*

Un programa Visual C++ de Microsoft es grande, normalmente un programa MFC puro no baja de 5MB sin incluir las librerías de enlace dinámico (DLLs) que la plataforma Windows necesita para ejecutar los programas Visual C++. En otras palabras, los programas C++ son completamente ejecutables en el ordenador en el que residen, lo que justifica que tengan un gran tamaño. Imaginemos que intentamos descargar todo eso como parte de una página Web para permitir que esa página haga algo interactivo en su ordenador.

Los programas Java, por el contrario, se construyen de forma diferente. El propio lenguaje Java está implementado como la máquina virtual de Java (JVM), que es la aplicación que actualmente ejecuta un programa Java. Cuando JVM se instala en un ordenador, éste puede ejecutar programas Java. Los programas Java, por lo tanto, no necesitan ser autosuficientes, y no tienen por qué incluir todo el código máquina que se ejecuta en el ordenador. En cambio, los programas Java son compilados creando *bytecodes* compactos y son estos *bytecodes* lo que JVM leę e interpreta para ejecutar el programa. Cuando descarga una *applet* Java de la red Internet, lo que realmente está descargando es un archivo de *bytecodes.*

De esta forma, su programa Java puede ser muy pequeño, ya que todo el código máquina necesario para ejecutarlo está ya en el ordenador de destino y no tiene que descargarse. Para distribuir Java entre una gran variedad de ordenadores, Sun sólo tuvo que rescribir JVM para que funcionara en esos ordenadores. Dado que su programa está almacenado en un archivo de *bytecode,* se ejecutará en cualquier ordenador en el que JVM esté instalado.

Aunque en un principio se suponía que los programas Java eran interpretados por JVM, es decir, ejecutados *bytecode* por *bytecode,* la interpretación podía ser un proceso lento. Por esta razón, Java 2 incorpora la compilación al instante *Just in Time* (JIT) en JVM. El compilador JIT realmente lee los *bytecodes* por secciones y los compila de forma interactiva en lenguaje máquina, por lo que el programa puede ejecutarse más rápido (todo el programa Java no se compila de una vez, ya que Java va realizando comprobaciones en tiempo de ejecución en varias secciones del código). Desde su punto de vista, esto quiere decir que su programa Java se ejecutará más rápido con el nuevo compilador JIT.

Utilizar *bytecodes* significa que los programas Java son muy compactos, lo que les hace ideales para descargarlos en la red Internet. Y otra ventaja a la hora de ejecutar tales programas con JVM, mayor que la descarga de programas, es la seguridad.

La seguridad del lenguaje Java

Cuando se ejecuta un programa, JVM puede monitorizar estrictamente lo que va ocurriendo, lo cual es importante para las aplicaciones de la red Internet. Fuera de toda duda, la seguridad ha llegado a ser un asunto extremadamente importante en la red Internet, y Java se ha lanzado a esa tarea. JVM puede ver todo lo que un programa hace, y si hay algo dudoso, como puede ser tratar de escribir en un archivo, puede prevenir esa operación. Eso sólo hace que, para la red Internet, Java sea más atractivo que C++, que no tiene tales restricciones.

Además se puede configurar la seguridad de Java de la forma que se desee, ofreciendo una solución muy flexible. Por ejemplo, como verá en este libro, ahora puede especificar, tomando como base programa por programa, aquellos privilegios que quiera dar para descargar código. Además, ahora puede firmar su código Java de forma que muestre de dónde procede para evitar modificaciones malévolas. Echaremos un vistazo a todo esto y más a lo largo de este libro.

Como puede observar, Java tiene una baza ganada en la red Internet: los programas Java son pequeños, seguros, independientes de la plataforma, orientados a objetos, y potentes. Además presentan otras características que gustan a los programadores. Los programas Java son robustos (lo que significa que son fiables y pueden gestionar bien los errores), con frecuencia son sencillos de escribir comparados con C++, son multihilo (pueden ejecutar un número de tareas al mismo tiempo, lo cual es útil cuando se quiere continuar haciendo otras cosas mientras se espera a que un archivo de datos se descargue, por ejemplo) y ofrecen un alto rendimiento. El resultado final es que se puede escribir un programa Java una vez y puede descargarse fácilmente y ejecutarse en todo tipo de máquinas, la receta perfecta para la red Internet. Esta es la razón por la que Java ha llegado tan alto.

Java no tiene como único objetivo a la red Internet; de hecho, hay dos tipos de programas Java, uno para el uso de la red Internet y otro para el uso en máquina local.

Programas Java

Los programas Java son de dos tipos principales: aplicaciones y *applets*. (En este libro utilizaré el término programa para referirme a *applets* y aplicaciones). Las *applets* son programas Java que pueden descargarse y ejecutarse como parte de una página Web, y son las que han hecho que Java sea tan popular. Por ejemplo, puede ver una *applet* en funcionamiento en la figura 1.1, donde la *applet* se está ejecutando en Microsoft Internet Explorer y visualiza un saludo.

La mayor parte de los navegadores Web han sido algunas veces lentos a la hora de implementar las versiones más recientes de Java, por lo que en este libro, usaré el visualizador de *applets* de Sun que viene con Java. Encontrará esta utilidad, llamada *appletviewer* (con una extensión según sea requerida por el sistema, como appletviewer.exe en Windows), en el directorio bin de Java junto con otras utilidades Java que usaremos en este y algunos de los siguientes capítulos. Puede ver el visualizador de *applets* funcionando en la figura 1.2, mostrando la misma applet que la figura 1.1.

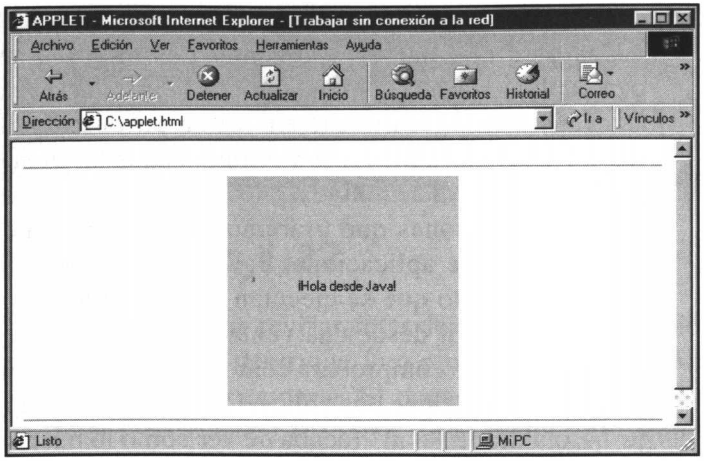


Figura 1.1. Una *applet* en funcionamiento.

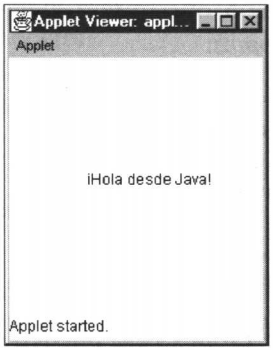


Figura 1.2. Una applet en funcionamiento con el visualizador de *applets* de Java.



Nota: ¿El que los fabricantes de navegadores Web sean lentos a la hora de actualizarlos con la última versión de Java significa que no pueda beneficiarse de las novedades sobre *applets* que incorpora Java 2, utilizando estos navegadores? No. Sun ha entrado en este tema y ha creado un *plug-in* para Netscape Navigator y Microsoft Internet Explorer que está integrado en la JVM de Java 2 como un *plug-in* de Netscape y un control ActiveX de Internet Explorer, respectivamente. Esto quiere decir que ahora puede ejecutar las últimas *applets* de Java en estos navegadores, al hacer cambios en la página Web la *applet* está ahí. Verá todo esto en el capítulo en el que se explican las *applets* formalmente, capítulo 6, después de cubrir suficientemente la sintaxis del lenguaje Java para empezar a crear *applets*.

Además de *applets* descargables, Java soporta aplicaciones que están diseñadas para ejecutarse localmente. Las aplicaciones Java funcionan como otras aplicaciones de ordenador, puede instalarlas y ejecutarlas en el suyo. Al estar instaladas localmente en vez de ser descargadas con una página Web, las aplicaciones tienen más privilegios que las *applets*, como es la capacidad para leer y escribir archivos.

Las aplicaciones que usaremos en los siguientes capítulos serán el tipo más sencillo de aplicaciones Java: aplicaciones consola. Son aplicaciones basadas en texto que se ejecutan desde la línea de comandos (en Windows esto quiere decir desde una ventana DOS), y pueden leer y visualizar texto. En este capítulo comprobará cómo tales aplicaciones funcionan. Por ejemplo, podrá decir que tiene una aplicación Java llamada app que visualiza el mensaje "¡Hola desde Java!" (acaba de ver cómo lo hace la *applet).* En este caso, la aplicación visualizará este mensaje en la pantalla. Ejecute la aplicación arrancando la máquina virtual de Java con el comando Java, pasándole el nombre de la aplicación que quiere ejecutar. En Unix, sería así, donde '%' es el *command prompt*:

```
%java app
¡Hola desde Java!
```

En Windows, sería así:

```
c:\>java app
¡Hola desde Java!
```

¿Las aplicaciones Java pueden ser gráficas? Realmente pueden serlo, y de hecho, la mayoría de ellas lo son. En este caso, la aplicación es responsable de arrancar su propia ventana (el visualizador de applets lo hará). En la figura 1.3 puede ver una aplicación ventana de Java.



Figura 1.3. Una aplicación ventana en ejecución.



Nota: introduciré aplicaciones ventana al mismo tiempo que las *applets*, en el capítulo 6, en el que empezaremos a trabajar con *Java Abstract Windowing Toolkit* (AWT), que es como se crean y manipulan las ventanas en Java. Si no puede esperar, echaremos un vistazo rápido a la creación de ventanas en este capítulo.

¿Es Java 2 o Java 1.2?

La última área que hay que revisar antes de empezar es la versión actual de Java. Puede que haya oído decir que Sun renombró Java 1.2 como Java 2, pero esto es sólo verdadero en parte. El paquete actual de Java, que será el que usemos, *Java Development Kit* (JDK), que incluye el compilador Java, JVM y otras utilidades, se llama oficialmente Java 2 JDK, versión 1.2. Por lo tanto, aunque me referiré a la versión actual de Java como Java 2, verá aún referencias a la versión 1.2; de hecho, se encontrará con alguna de estas referencias en la siguiente sección.

Para una visión general ya es suficiente; es hora de empezar a crear programas Java y de ver cómo avanza.

Adquirir e instalar Java

El gran jefe le llama, como es usual, en el último minuto. Tiene 20 minutos para escribir una nueva página Web que permita a los usuarios tener una visión general de los productos de su compañía. ¿Qué va a hacer? Conociendo lo bien que funciona Java en casos como este, selecciona Java como lenguaje para realizar esa tarea. Por supuesto, se ha asegurado de tenerlo antes de poder utilizarlo.

Es hora de descargar e instalar Java, lo que significa descargar *Java Development Kit* (JDK), que está disponible en http://java.sun.com/products/jdk/1.2/ (1.2 se refiere a la versión de JDK, que ahora se llama Java 2 *Development Kit,* versión 1.2.2 en el momento de escribir este manual).

Después de descargar JDK, generalmente como un paquete ejecutable autoinstalable, seguir las instrucciones de instalación en el sitio Web java.sun.com; por ejemplo, las instrucciones de instalación en Windows están en http://java.sun.com/products/jdk/1.2/install-windows.html, y para Solaris se encuentran en http://java.sun.com/products/jdk/1.2/install-solaris.html.

Estaría encantado de proporcionar aquí las instrucciones de instalación, pero esa es una de las mayores trampas en las que un libro de Java puede caer,

incluso uno que haya sido diseñado para ser lo más completo posible. Llevo escribiendo sobre Java desde sus orígenes y se ha demostrado que las instrucciones de instalación son muy volátiles. Como esas instrucciones cambian, las que incorporé en los libros anteriores se quedaron obsoletas al instante, dando lugar a gran cantidad de llamadas y cartas. Por esa razón, lo mejor que se puede hacer es ver cómo Sun quiere que se instale JDK, y por consiguiente, debería acudir a las instrucciones de instalación que se encuentran en el sitio Web de Java. El proceso de instalación se ha ido haciendo cada vez más fácil en las diferentes versiones y ahora sólo consiste en ejecutar el archivo que se ha descargado.

Algo que se debería hacer, como indican las instrucciones de instalación de Sun, es asegurarse de que su máquina es capaz de localizar las herramientas de Java, incluyendo el compilador. Para hacer eso, verificar que el directorio bin de Java está en la ruta de acceso de su ordenador. Por ejemplo, en Windows 95/98, el directorio bin es c:\jdk1.2.2\bin para JDK 2, versión 1.2.2, y basta con añadir una línea como esta (asegúrese de que están incluidos en *la ruta de acceso* todos los directorios que quiere utilizar) en el archivo autoexec.bat:

```
SET PATH=C:\WINDOWS;C:\JDK1.2.2\BIN
```

Además, debe volver a arrancar su ordenador para que los cambios tengan efecto. Cuando el directorio bin esté en la ruta de acceso, podrá utilizar las herramientas de Java directamente desde la línea de comandos, en lugar de tener que invocarlas con toda la ruta de acceso cada vez.

¿Qué ocurre con CLASSPATH?

Los veteranos en Java se preguntarán por la variable de entorno llamada CLASSPATH cuando instalen Java2. La variable CLASSPATH, como veremos pronto en este capítulo, dice a Java donde encontrar los archivos *bytecode* compilados, tanto los que creó como los del sistema que vienen con JDK. CLASSPATH ha sido el foco de grandes confusiones al trabajar con Java y estoy contento de decir que Sun ha facilitado las cosas.

Cuando instale JDK, ya no tiene que preocuparse de poner la variable CLASSPATH, porque JDK sabrá dónde buscar sus propias librerías. Sin embargo, si quiere buscar otros archivos *bytecode* creados al compilar un archivo, tendrá que poner la variable CLASSPATH. Verá cómo hacer esto cuando discutamos la compilación de programas en este capítulo (hay dos formas de indicar al compilador de Java dónde buscar los archivos *bytecode*

que quiere encontrar: poniendo la variable de entorno CLASSPATH y usando la opción del compilador -classpath).

En Java 2 hay muchas novedades. Sin embargo, Java 2 también tiene algunas cosas obsoletas, en Java llamadas *censuradas*, y es importante saber lo que es obsoleto y lo que es nuevo. De hecho, echaremos un vistazo a lo que era nuevo en Java 1.1 para servir de utilidad a los programadores que proceden de Java 1.0. Veamos este tema a continuación.

¿Cuáles son las novedades de Java 1.1?

El programador novato (PN) comienza, como es usual, buscando ayuda. "He utilizado Java 1.0", dice el PN, "y estaba pensando en actualizarlo a Java 1.1". "Bien", responderá , "la versión actual es la 2". El PN ignora eso y pregunta, "¿Cuáles son las novedades de Java 1.1?" "Bien", dirá, "muchas cosas, tome una silla y las iremos viendo".

Probablemente el mayor cambio entre Java 1.0 y 1.1 fue la forma en que los programas Java gestionaban los eventos. Un evento ocurre cuando el usuario ejecuta alguna acción significativa en la interfaz de usuario, como pulsar un botón, mover el ratón o pulsar una tecla (veremos todo sobre las viejas y nuevas técnicas de gestión de eventos en el capítulo 6). La nueva técnica en Java 1.1 y 2 utiliza el modelo de eventos *delegado*, y es bastante diferente de cómo se hacía en Java 1.0. Tan diferente, de hecho, que si intenta ejecutar un programa Java que usa el modelo de evento *delegado* en un navegador Web que sólo soporta el modelo de evento de Java 1.0, el proceso se detendrá. (En cambio, todavía podrá usar las técnicas antiguas de gestión de eventos en Java 1.1 y 2, pero el rendimiento de los programas se verá degradado).

Hubo muchos cambios en Java 1.1. A continuación se enumeran los más importantes (observar que algunos de estos temas no tienen mucho sentido ahora pero estarán más claros a lo largo del libro):

- Mejoras en *Abstract Windowing Toolkit* (AWT). Java 1.1 soportaba la impresión, más rapidez en los desplazamientos, menús emergentes, el portapapeles, un modelo de eventos basado en la delegación, mejoras en las imágenes y gráficos y más temas. Además, era más rápido que antes (algo que los programadores de Java podían apreciar de forma definitiva).
- Archivos JAR. Los archivos JAR (archivo Java) permiten empaquetar un número de archivos, comprimiéndolos para reducir su tamaño y

permitiendo la descarga de muchos archivos a la vez. En un archivo JAR se pueden poner muchas *applets* y los datos que necesiten, de forma que se descargue mucho más rápido.

- Internacionalización. Se pueden desarrollar *applets* locales, utilizando caracteres UNICODE, mecanismo local, soporte de mensajes locales, fecha sensible a la localidad, hora, zona horaria y más.
- *Applets* firmadas y firmas digitales. Se pueden crear aplicaciones Java con firmas digitalizadas. Una firma digital permite a los usuarios dar marcha atrás en el caso de que algo vaya mal. Esto forma parte de las nuevas precauciones de seguridad de la red mundial (*World Wide Web).*
- Método remoto de llamada (RMI). Permite que los objetos Java tengan métodos que son llamados desde el código Java que se ejecuta en otras sesiones.
- Objetos en serie de objetos. Permite almacenar objetos y gestionarlos con flujos binarios de entrada/salida. Permite almacenar copias de los objetos que se van a serializar, y además es la base de la comunicación entre objetos incluidos en RMI.
- Reflexión. Permite al código Java examinar la información sobre los métodos y constructores de clases cargadas, así como hacer uso de esos métodos reflejados y constructores.
- Clases internas. Hay clases encerradas en otras clases y el uso de las clases internas hace que sea más fácil crear clases adaptadoras. Una clase adaptadora es una clase que implementa una interfaz que es requerida por un API (interfaz de programación de aplicaciones). Utilizando las clases adaptadoras se facilita la gestión de eventos, como veremos más tarde.
- Interfaz de un nuevo método nativo de Java. El código nativo es un código que se escribe específicamente para una máquina particular. La escritura y la llamada a código nativo puede mejorar significativamente la velocidad de ejecución. Java 1.1 incluía un nuevo método nativo de interfaz.
- Clases de tipo *byte* y *short*. Los valores de tipo *byte* y *short* pueden ser gestionados como números "envueltos" cuando se usan las nuevas clases de tipo *Byte* y *Short*.
- *JavaBeans*. Son componentes Java que pueden conectarse con otros programas Java. Es una nueva característica muy potente.

La mayoría de los métodos de Java 1.0 han quedado obsoletos con Java 1.1, y están marcados como censurados en la documentación de Java 1.1. Además, el compilador Java da un aviso cuando se compila el código que utiliza algo censurado. Ver el siguiente tema para más información.

¿Qué está censurado en Java 1.1?

"¡Eh!," dice el programador novato, "actualizo a Java 1.1 ahora, y mi código es erróneo. Por ejemplo, ahora estoy teniendo errores cuando utilizo la clase Date de Java". "Eso es porque gran parte de la clase Date fue censurada en Java 1.1," dice.

La mayor parte de las características de Java 1.0 se censuraron en Java 1.1, muchas más que la lista que aquí mostramos. Puede encontrar una lista de todo lo que se censuró en Java 1.1 en http://java.sun.com/products/jdk/1.1/docs/relnotes/deprecatedlist.html. Sin embargo, observe que esta lista no le será muy útil si usa Java 2, como haremos en este libro; en cambio vea el tema "¿Qué está censurado en Java 2?", que viene a continuación.

¿Cuáles son las novedades de Java 2?

"De acuerdo", dice el programador novato, "usted gana. Está claro que, debería actualizar a Java 2, no a Java 1.1. Entonces, ¿cuáles son las novedades de Java 2?" "Muchas cosas", dice. "Mejor tomemos un café". "Justo cuando ya me había habituado a Java 1.1", se queja el programador novato.

A continuación verá las novedades de Java 2 (observe que como en la lista mostrada en "¿Cuáles son las novedades de Java 1.1?", no tiene que estar familiarizado ahora con los conceptos, pero los verá más tarde en este libro):

- Mejoras en la seguridad. Ahora, cuando el código está cargado, se asignan permisos basándose en las políticas de seguridad que actualmente tienen efecto. Cada permiso indica que se tiene un acceso permitido a un recurso particular (como acceso de lectura y escritura a un archivo o directorio concreto, acceso para conectarse a un *host* o puerto, etc.). La política que especifica qué permisos están disponibles para el código de varios firmantes/lugares, puede ser inicializada desde un archivo externo configurable. A menos que un permiso sea garantizado explícitamente, no se puede acceder al recurso que está protegido por ese permiso.

- *Swing* (JFC). Forma parte de las clases Java Foundation (JFC). Implementa un nuevo conjunto de componentes GUI de tipo *pluggable*. *Swing* está implementado en Java puro y está basado en JDK 1.1 *Lightweight UI Framework*. El carácter *pluggable* permite diseñar un conjunto sencillo de componentes GUI que pueden automáticamente tener la apariencia de cualquier plataforma (por ejemplo, Windows, Solaris y Macinstosh).
- Java 2D (JFC). El API Java 2D es un conjunto de clases para gráficos 2D e imágenes. Engloba el conjunto de líneas, texto e imágenes en un modelo sencillo y extenso.
- Accesibilidad (JFC). Por medido del API de accesibilidad de Java, los desarrolladores pueden crear aplicaciones Java que interactúen con tecnologías asistenciales, tales como lectores de pantalla, sistemas de reconocimiento de voz y terminales Braille.
- Arrastrar el cursor y soltar (JFC). Permite la transferencia de datos entre Java y aplicaciones nativas, entre aplicaciones Java y en el interior de una aplicación Java sencilla.
- Colecciones. El API de colecciones Java es un armazón unificado para representar y manipular colecciones Java (verá más sobre ellas más tarde), permitiéndoles ser manipulados independientemente de los detalles de su representación.
- *Framework* de extensiones Java. Las extensiones son paquetes de clases Java (y el código nativo asociado) que los desarrolladores de aplicaciones pueden usar para heredar el núcleo de la plataforma Java. El mecanismo de herencia permite que la máquina virtual de Java (JVM) utilice las clases heredadas de la misma forma que JVM utiliza las clases del sistema.
- Mejoras en *JavaBeans*. Java 2 proporciona a los desarrolladores un método estándar para crear componentes y aplicaciones *JavaBeans* más sofisticados que ofrecen a sus clientes una integración limpia con el resto del entorno de ejecución, como el escritorio del sistema operativo o del navegador.
- *Framework* para las entradas. Permite que todos los componentes de los editores de texto reciban textos en japonés, chino o coreano a través de los métodos estándar de entrada.
- Paquete de identificación de versiones. Las aplicaciones y las *applets* pueden identificar (en tiempo de ejecución) la versión de un entorno de ejecución Java específico, JVM y paquete de clases.

- Mejoras RMI. El método remoto de llamada (RMI) tiene varias mejoras, incluyendo la activación remota de objetos, que introduce el soporte de objetos remotos y la activación automática de objetos, así como configurar los tipos de *socket*, que permite a un objeto remoto especificar el tipo de *socket* configurado que RMI usará para llamadas remotas a ese objeto. RMI, sobre un transporte seguro (como SSL), puede ser soportado usando tipos de *socket* configurados.
- Mejoras en la serialización. La serialización ahora incluye un API que permite que los datos serializados de un objeto sean especificados independientemente de los campos de la clase. Esto permite que los campos de datos serializados se escriban y se lean desde un flujo de datos usando las técnicas existentes (esto asegura la compatibilidad con los mecanismos de escritura y lectura establecidos por defecto).
- Objetos referencia. Un objeto referencia encapsula una referencia a algún otro objeto para que la referencia, por sí misma, pueda ser examinada y manipulada como cualquier otro objeto. Los objetos referencia permiten a un programa mantener una referencia a un objeto que no impide al objeto ser regenerado por el reciclador, el cual gestiona la memoria.
- Mejoras en el audio. Las mejoras en el audio incluyen una nueva máquina de sonido y soporte para el audio en aplicaciones y en *applets*.
- Java IDL. Java IDL incorpora la capacidad de CORBA (*Common Object Request Broker Architecture)* a Java, proporcionando la interoperabilidad y conectividad basada en estándares. Java IDL permite que las aplicaciones Java Web distribuidas invoquen de forma transparente operaciones o servicios de red remotos usando el estándar de la industria OMG IDL (*Object Management Group Interface Definition Language)* e IIOP (*Internet Inter.-ORB Protocol)* definido por el Grupo de Gestión de Objetos.
- Mejoras en JAR. Estas mejoras incluyen la funcionalidad añadida a la herramienta JAR de la línea de comandos para crear y actualizar archivos JAR firmados. Hay además nuevas APIs estándares para leer y escribir archivos JAR.
- Mejoras en JNI. La interfaz nativa Java es una interfaz de programación estándar para escribir métodos nativos de Java y embeber la máquina virtual de Java en aplicaciones nativas. El primer objetivo es la compatibilidad binaria de librerías de método nativo a través de todas las implementaciones de la máquina virtual Java en una plataforma dada.

Java 2 extiende el JNI incorporando nuevas características en la plataforma Java.

- JVMDI. Una nueva interfaz de *debugger* para la máquina virtual de Java proporciona ahora servicios de bajo nivel para el *debug*. La interfaz para estos servicios es la interfaz *Debugger* de la máquina virtual de Java (JVMDI).
- Mejoras JDBC. La conectividad Java de bases de datos (JDBC) es una interfaz estándar de acceso a base de datos que proporciona un acceso uniforme a un amplio abanico de bases de datos relacionales. JDBC además proporciona una base común en la que se pueden construir interfaces y herramientas de alto nivel. Java 2 incluye JDBC y el enlace JDBC-ODBC.

En Java 2 también es nuevo el Java *plug-in*. Este software es un producto que permite a los usuarios dirigir applets de Java o componentes *JavaBeans* para que se ejecuten en el entorno Java (JRE) de Sun, en lugar del entorno de ejecución del navegador Web que viene por defecto con Java. Esto se tratará en el capítulo 6.

¿Qué se censuró en Java 2?

"¡Eh!", dice el programador novato, "ya he actualizado a Java 2 y mi código sigue siendo defectuoso, no consigo que la parte de mi código en la que utilizo multihilos funcione del todo". "Eso es porque no puedes utilizar los métodos *resume*, *suspend* y *stop* de la clase *Thread* de Java,", dice. "Se han censurado". "¡Vaya!," protesta el programador novato.

Uno de los cambios más importantes de Java 2 es que no se trabaja con hilos como se hacía en Java 1.1; verá las nuevas formas en el capítulo que trata multihilos más tarde en el libro. Para una lista completa de lo que se ha censurado en Java 2, ir a C:\jdk1.2.2\docs\api\deprecated-list.html. Haré mención a alguno de los temas censurados más importantes en Java 2 a lo largo del libro.

Escribir código: creación de archivos de código

El coordinador del equipo de diseño (CED) le llama para felicitarle por la instalación de Java. Usted acepta los agradecimientos. "¿Qué programas se han escrito?" pregunta el CED. "Hmm", piensa. "¿Programas?"

Los programas Java son archivos de texto planos formados por instrucciones y declaraciones Java, y empezaremos a investigarlos en el siguiente apartado. Para crear un programa Java, debería tener un editor o procesador de texto que permita guardar archivos en formato de texto plano.

Guardar texto en formato de texto plano es una ejecución sencilla quela mayor parte de los procesadores de texto soportan. Podría tener problemas con procesadores de texto como Microsoft Word, por ejemplo, aunque puede guardar archivos de texto con Word ejecutando el comando Guardar como del menú Archivo. La regla general es que si al editar un archivo desde la línea de comando no se ve ningún carácter que no sea alfanumérico suelto, se trata de un archivo de texto plano. La prueba real, por supuesto, es que el compilador de Java, que traduce su programa en un archivo de *bytecode*, pueda leer e interpretar dicho programa.

Además, los programas deben estar almacenados en archivos que tengan extensión "java". Por ejemplo, si está escribiendo una aplicación llamada app, debería guardar el programa Java en un archivo llamado app.java. Pase este archivo al compilador de Java para crear el archivo de *bytecode*, como verá en las siguientes páginas.

Ya tenemos la selección del editor o procesador de textos. Ahora, ¿qué pasa con la escritura del código?

Escribir código: conocer las palabras reservadas de Java

El programador novato aparece y dice, "Java se está comportando de forma graciosa, quiero llamar a una variable "*public*" y me está dando muchos problemas". "Eso es porque *public* es una de las palabras que Java reserva como parte del lenguaje Java", le dice. "¡Vaya!", dice el programador novato.

Cuando esté escribiendo código Java, debería saber que Java reserva ciertas palabras clave como parte del lenguaje. No hay muchas, de cualquier modo. A continuación se muestran (trataré estas palabras clave a lo largo del libro):

- *abstract*: Especifica la clase o método que se va a implementar más tarde en una subclase.
- *boolean*: Tipo de dato que sólo puede tomar los valores verdadero o falso.

- *break:* Sentencia de control para salirse de los bucles.
- *byte*: Tipo de dato que soporta valores en 8 bits.
- *byvalue*: Reservada para uso futuro.
- *case*: Se utiliza en las sentencias *switch* para indicar bloques de texto.
- *cast*: Reservada para uso futuro.
- *catch*: Captura las excepciones generadas por las sentencias *try.*
- *char*: Tipo de dato que puede soportar caracteres *Unicode* sin signo en 16 bits.
- *class*: Declara una clase nueva.
- *const*: Reservada para uso futuro.
- *continue*: Devuelve el control a la salida de un bucle.
- *default*: Indica el bloque de código por defecto en una sentencia *switch.*
- *do*: Inicia un bucle *do-while.*
- *double:* Tipo de dato que soporta números en coma flotante, 64 bits.
- *else*: Indica la opción alternativa en una sentencia *if.*
- *extends*: Indica que una clase es derivada de otra o de una interfaz.
- *final*: Indica que una variable soporta un valor constante o que un método no se sobrescribirá.
- *finally*: Indica un bloque de código en una estructura *try – catch* que siempre se ejecutará.
- *flota*: Tipo de dato que soporta un número en coma flotante en 32 bits.
- *for*: Utilizado para iniciar un bucle *for.*
- *future:* Reservada para uso futuro.
- *generic*: Reservada para uso futuro.
- *goto*: Reservada para uso futuro.
- *if*: Evalúa si una expresión es verdadera o falsa y la dirige adecuadamente.
- *implements*: Especifica que una clase implementa una interfaz.
- *import*: Referencia a otras clases.

- *inner*: Reservada para uso futuro.
- *instanceof*: Indica si un objeto es una instancia de una clase específica o implementa una interfaz específica.
- *int*: Tipo de dato que puede soportar un entero con signo de 32 bits.
- *interface*: Declara una interfaz.
- *long*: Tipo de dato que soporta un entero de 64 bits.
- *native*: Especifica que un método está implementado con código nativo (específico de la plataforma).
- *new*: Crea objetos nuevos.
- *null*: Indica que una referencia no se refiere a nada.
- *operator*: Reservado para uso futuro.
- *outer*: Reservado para uso futuro.
- *package*: Declara un paquete Java.
- *private*: Especificador de acceso que indica que un método o variable sólo puede ser accesible desde la clase en la que está declarado.
- *protected*: Especificador de acceso que indica que un método o variable sólo puede ser accesible desde la clase en la que está declarado (o una subclase de la clase en la que está declarada u otras clases del mismo paquete).
- *public*: Especificador de acceso utilizado para clases, interfaces, métodos y variables que indican que un tema es accesible desde la aplicación (o desde donde la clase defina que es accesible).
- *rest*: Reservada para uso futuro.
- *return*: Envía control y posiblemente devuelve un valor desde el método que fue invocado.
- *short*: Tipo de dato que puede soportar un entero de 16 bits.
- *static*: Indica que una variable o método es un método de una clase (más que estar limitado a un objeto particular).
- *super*: Se refiere a una clase base de la clase (utilizado en un método o constructor de clase).
- *switch*: Sentencia que ejecuta código basándose en un valor.

- *synchronized*: Especifica secciones o métodos críticos de código multihilo.
- *this*: Se refiere al objeto actual en un método o constructor.
- *throw*: Crea una excepción.
- *throws*: Indica qué excepciones puede proporcionar un método.
- *transient*: Especifica que una variable no es parte del estado persistente de un objeto.
- *try*: Inicia un bloque de código que es comprobado para las excepciones.
- *var*: Reservado para uso futuro.
- *void*: Especifica que un método no devuelve ningún valor.
- *volatile*: Indica que una variable puede cambiar de forma asíncrona.
- *while*: Inicia un bucle *while*.

Escribir código: crear una aplicación

El gran jefe llega y dice, "Ya puede escribir Java, ¿no? ¡Hágame una demostración!" Usted se va hacia la terminal e inmediatamente su mente se queda en blanco. ¿Qué va a escribir?

A continuación hay un ejemplo de aplicación Java que se desarrollará en las siguientes secciones del libro, así como los estados de compilación y ejecución. Escriba este código en un archivo llamado app.java:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Si es nuevo en Java, esto puede que le resulte extraño. La idea es que esta aplicación visualice el texto "¡Hola desde Java!" cuando se compile y se ejecute. Por ejemplo, así sería en una ventana DOS desde Windows:

```
C:\>java app
¡Hola desde Java!
```

No se trata de uno de los programas más significativos, pero sí es bueno para empezar. Veamos este programa línea por línea.

public class app

Esta es la primera línea de app.java:

```
public class app
{



}
```

Esta línea indica que estamos creando una clase de Java nueva llamada app. Después de que esta clase la transformemos en *bytecodes*, la máquina virtual de Java podrá crear objetos de esta clase y ejecutarlos. Aprenderá todo sobre las clases en el capítulo 4; este código es sólo para empezar con la programación Java.

Observe la palabra clave *public* en el código anterior. Esta palabra es un especificador de acceso, sobre la que aprenderá más en los capítulos 4 y 5. El especificador de acceso *public* indica que esta clase está disponible en cualquier parte del programa que la utilice.

Observe además que si construye una clase pública, Java le obliga a dar un nombre al archivo. Es decir, sólo puede tener una clase pública en un archivo con extensión ".java". La razón de esto es que el compilador de Java traduce el archivo de extensión ".java" en un archivo *bytecode* con la extensión ".class", lo que significa que app.java se convertirá en app.class, y si JVM necesita la clase app, sabrá cómo mirar en el archivo app.class. Dado que JVM utiliza el nombre del archivo para determinar las clases públicas que hay en él, sólo se puede tener una en cada archivo. Por esa razón, el código para la clase app debe estar en un archivo llamado app.java (observe que Java es bastante particular en esto, y el uso de mayúsculas hay que tenerlo en cuenta).

La implementación de la clase que estamos definiendo ahora, estará entre llaves:

```
public class app
{



}
```

Java siempre encierra bloques de código entre llaves, es decir, '{' y '}'. Como verá en el capítulo 4, el código de un bloque tiene su propio alcance (su visibilidad para el resto del programa). A partir de ahora, continuaremos construyendo nuestra aplicación siguiendo con la siguiente línea de código.

public static void main[String[] args)

Esta es la siguiente línea de código de nuestra aplicación:

```
public class app
{
    public static void main(String[] args)
    {

    }
}
```

Aquí estamos creando un método de la clase app. Un método en la programación orientada a objetos es como una función o subrutina en la programación estándar, un bloque de código al que se le puede pasar el control y que puede devolver un valor. Los métodos permiten manejar fácilmente el código en una unidad funcional sencilla; cuando llama a un método, la máquina virtual de Java ejecuta el código de ese método.

Los métodos serán tratados formalmente en el capítulo 4, pero aquí la idea es que estamos creando un método llamado *main*, que es el método que la máquina virtual de Java busca cuando inicia una aplicación (las applets no tienen método *main*). Cuando encuentra el método *main*, JVM le pasa control, y nos situamos en la parte del código que queremos ejecutar de este bloque de código del método.

Antes de continuar hay varias cosas que observar. El método *main* debe declararse con el especificador de acceso *public*, lo que quiere decir que puede ser llamado desde fuera de su clase. También debe declararse como *static*, que significa, como veremos en el capítulo 4, que *main* es un método de una clase, no un método de un objeto. Cuando se termine de ejecutar, no debe devolver ningún valor, por lo cual usamos la palabra *void* en este código (en otras palabras, un valor de retorno de tipo *void* significa que actualmente no devuelve valor). Finalmente, observe el argumento entre paréntesis que sigue a la palabra main: String[] args. Aparece una lista de argumentos entre paréntesis en la declaración de un método para indicar que los valores se le pasan al método y que el código del método puede usarlo. En este caso, estamos indicando que a *main* se le pasa un *array* cuyos elementos son cadenas de caracteres, llamado args. Estos elementos son valores que se pasan desde la línea de comandos cuando se inicia la aplicación; por ejemplo, si escribe "java app Hola ahí", entonces "Hola" y "ahí" serían las dos cadenas del *array* args. Todos los detalles se mostrarán en el capítulo 4. Dado que no usaremos ningún argumento desde la línea de comandos en esta aplicación, no usaremos args en el código del método *main*.

Esta línea de código, entonces, inicia el método *main*. Todo el trabajo de este método es visualizar el texto "¡Hola desde Java!", lo que se realizará en la siguiente línea de código.

System.out.println("¡Hola desde Java!");

El método *main* tiene una línea de código:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Esta es la línea de código que hace todo el trabajo. En este caso, estamos usando el código que los programadores de Sun ya han creado para visualizar el texto "¡Hola desde Java!". En particular la clase *System* del paquete java.lang. En Java, las librerías de clases se llaman paquetes y el paquete java.lang está en todo programa Java, lo que quiere decir que no hay que hacer nada especial para utilizarlo, a diferencia de otros paquetes Java. La clase *System* del paquete java.lang incluye un campo (es decir, un miembro de datos de la clase) llamado *out*, y este campo, a su vez, tiene un método llamado *println*, que hace que se visualice el texto.

Para referirse al campo *out* de la clase *System*, usamos la terminología System.out. Para utilizar el método println del campo *out*, usamos la terminología System.out.println. Para imprimir el texto "¡Hola desde Java!", se lo pasamos a System.out.println cerrándolo entre comillas.

Observe además que esta línea de código termina con un punto y coma (';'). Esto es algo que Java ha heredado de C y C++ (de hecho, Java ha heredado muchas cosas de C y C++), y casi todas las sentencias de Java se terminan con punto y coma. Si no está acostumbrado a ello, lo conseguirá rápidamente, ya que el compilador de Java rechaza la traducción a *bytecodes* hasta que el punto y coma esté en su sitio.

Por lo tanto, hemos creado una nueva aplicación y la hemos almacenado en un archivo llamado app.java. ¿Cuál es el siguiente paso? Ejecutarlo. Echemos un vistazo al siguiente tema.

Compilación

El gran jefe, mascando un cigarro, permanece de pie detrás de usted mientras guarda la nueva aplicación Java en un archivo. "Hmm", dice el gran

jefe, nada impresionado. "¿Y ahora?" "Ahora", dice, "tenemos que compilar el programa y ya podremos ejecutarlo". "De acuerdo," dice el gran jefe. "Sorpréndame".

Para traducir un programa Java a un archivo *bytecode* que la máquina virtual de Java pueda utilizar, usará el compilador de Java, que se llama javac (por ejemplo, en las máquinas Windows, este programa se llamará javac.exe, que está en el directorio bin de Java). A continuación se indica cómo usará javac en general:

```
javac [opciones] [archivos fuente] [@archivos]
```

Argumentos de javac:

- opciones: Opciones desde la línea de comandos.
- archivos fuente: Uno o más archivos que se van a compilar (como app.java).
- @archivos: Uno o más archivos de la lista de archivos fuente.

Para compilar app.java, usaremos este comando:

```
C:\javac app.java
```

El compilador de Java, javac, toma el archivo app.java y (suponiendo que no haya errores) lo compila, traduciéndolo y creando un nuevo archivo llamado app.class. Si hubiera errores, el compilador de Java nos avisará de que hay errores incluyendo qué línea de código está mal, como en este caso en el que hemos olvidado el método *println* e intentamos usar uno llamado *printline*:

```
C:\javac app.java
app.java:5: Method printline(java.lang.String) not found in class
java.io.Print
Stream.
  System.out.printline("¡Hola desde Java!");
1 error
```

Cuando app.java está compilado sin errores, el nuevo archivo, app.class, contiene todo lo que la máquina virtual de Java necesitará para crear objetos de la clase app. Ya hemos creado app.class. Ahora, ¿cómo lo ejecutamos en JVM? Veamos el siguiente punto.

Compilación: utilizando opciones en la línea de comandos

"Hmm", dice el programador novato, "tengo un problema. Me gusta guardar todos mis archivos con extensión ".class" en el mismo directorio, pero

algunas veces se me olvida copiar las nuevas versiones de esos archivos en ese directorio". "Hay una opción del compilador que es perfecta para eso: la opción –d. Usando esa opción, puede situar los archivos creados por el compilador en el directorio de destino que quiera". "Entonces, sólo tengo que acordarme de utilizar esa opción...," dice el programador.

Hay un buen número de opciones que se pueden usar con javac. Por ejemplo, indicaremos cómo usar la opción –d para situar el archivo app.class en el directorio temp., que ya existe y que, en este caso, es un subdirectorio del directorio actual:

```
javac -d Temp. app.java
```

A continuación se muestra la lista de opciones de javac. Observe que las opciones que empiezan con –X (llamadas opciones no estándar) están marcadas de esa forma por Sun ya que pueden cambiar en el futuro:

- -classpath ruta de acceso. Establece la ruta de acceso del usuario, sobrescribiéndola en la variable de entorno CLASSPATH. Si no se especifica CLASSPATH ni –classpath, la ruta de acceso es el directorio actual. Observe que si no se usa la opción –sourcepath, la ruta de acceso se busca tanto para los archivos fuentes como para los archivos de clase.
- -d directorio. Fija el directorio de destino de los archivos de clase. Para los lectores que sepan qué son los paquetes Java, si una clase forma parte de un paquete, javac pone el archivo de clase en un subdirectorio que refleja el nombre del paquete, creando los directorios necesarios. Por ejemplo, si especifica –d c:\clases y la clase se llama com.package1.Class1, el archivo de clases se llama c:\clases\com\package1\Class1.class. Si no se especifica la opción –d, javac pone el archivo de clases en el mismo directorio que el archivo fuente. Observe que el directorio indicado por –d no se añade automáticamente a su ruta de acceso de clases.
- -deprecation. Muestra una descripción de cada uso o sobrescritura de un miembro o clase censurado. (Sin –deprecation, javac sólo muestra los nombres de los archivos fuente que usan o sobrescriben miembros o clases censurados).
- -encoding. Fija el nombre codificado del archivo fuente. Si no se especifica –encoding, se usará el convertidor por defecto de la plataforma.
- -g. Genera todo la información de *debugging*, incluyendo variables locales. Por defecto, sólo se genera el número de línea e información del archivo fuente.

- -g:none. Hace que el compilador no genere ningún tipo de información de *debugging*.
- -g:(lista de palabras reservadas). Sólo genera algunos tipos de información de *debugging*, especificados por la lista de palabras reservadas separadas por coma. Las palabras válidas son source (información de *debugging* del archivo fuente), lines (información de *debugging* del número de línea) y vars (información de *debugging* de las variables locales).
- -nowarn. Deshabilita todos los mensajes de aviso.
- -O. Optimiza el código para la ejecución en términos de hacer más rápido el tiempo de ejecución. Observe que utilizar la opción –O puede ralentizar la compilación, producir archivos de clase más grandes y hacer que el programa sea difícil de poner a punto. Observe que antes de tener JDK 1.2, las opciones –g y –O no podían usarse juntas. Con JDK 1.2, puede combinar –g y –O, pero puede obtener resultados inesperados, como pérdida de variables o de código.
- -sourcepath ruta de acceso de fuentes. Especifica la ruta de acceso de las fuentes para las definiciones de clases o de interfaces. Al igual que con la ruta de acceso de clases, las entradas se separan por punto y coma (';') y pueden ser directorios, archivos Java (con extensión ".jar") o archivos zip. Si usa paquetes, el nombre local de la ruta de acceso dentro del directorio o del archivo debe reflejar el nombre del paquete, como verá más tarde. Observe que las clases buscadas en la ruta de acceso de clase serán recompiladas automáticamente si sus archivos de código fuente no se encuentran.
- -verbose. Crea la salida "verbose". Incluye información sobre cada clase cargada y de cada archivo fuente compilado.
- -X. Visualiza información sobre las opciones no estándares.
- -Xdepend. Busca todas las clases que pueden ser utilizadas por los archivos fuente más recientes para recompilar. Esta opción descubrirá clases que necesitan ser recompiladas, pero puede ralentizar enormemente el proceso de compilación.
- -Xstdout. Envía mensajes del compilador a System.out. Por defecto, los mensajes del compilador van a System.err, que verá más tarde.
- -Xverbosepath. Describe las rutas de acceso y las extensiones estándar que fueron localizadas para buscar archivos fuente y de clases.

- -Joption. Usa esta opción para pasar una opción al lanzador de Java llamado por javac. Por ejemplo, -J-Xms64m fija la memoria de inicio a 64 MB. Aunque esta opción no empieza con –X, no es una opción estándar de javac. Es una convención para pasar opciones a las aplicaciones escritas en Java que se están ejecutando en JVM.

Opciones de compilación cruzada

Las opciones de compilación cruzada se consideran un tema avanzado de Java; javac soporta la compilación cruzada, en la que las clases se compilan mediante un proceso autosuficiente (por defecto) y la implementación de extensión de clases. Se debe utilizar -bootclasspath y -extdirs cuando se esté hablando de compilación cruzada. A continuación se indican las opciones de compilación cruzada:

- -target versión. Genera los archivos de clase que trabajará en JVM con la versión indicada. Por defecto se generan archivos de clase que son compatibles con JVM 1.1 y 1.2. Las versiones soportadas por javac en JDK 1.2 son 1.1 (asegura que los archivos de clase generados serán compatibles con JVM 1.1 y 1.2; esto es por defecto) y 1.2 (genera archivos de clase que se ejecutarán en JVM 1.2 pero no en JVM 1.1).
- -bootclasspath ruta de acceso de arranque. Realiza la compilación cruzada contra el conjunto de clases de arranque indicadas. Al igual que con la ruta de acceso de clases, estas entradas están separadas por punto y coma (';') y pueden ser directorios, archivos con extensión".jar" o archivos zip.
- -extdirs directorios. Realiza la compilación cruzada contra los directorios indicados; los directorios están separados por punto y coma (';'). Cada archivo con extensión ".jar" de los directorios indicados es localizado automáticamente por los archivos de clase.

Compilación: revisión de los métodos censurados

"Bueno", dice el programador, "¿cómo puedo cambiar todos los métodos censurados para que sean correctos? Ahora que ya estoy en Java 2, no sé lo que es obsoleto y lo que no". "Eso es fácil", responde. "El compilador de Java, javac, le dirá si está usando un método que ya ha sido abandonado.

Mejor aún, puede usar la opción –deprecation para asegurarse de tener todos los detalles".

La opción –deprecation es buena, y es la opción estándar que utilizo para asegurarme de que estoy evitando los métodos obsoletos. Supongamos que tiene un programa de 200 líneas, y que cuando lo intenta compilar, javac le devuelve este resultado:

```
C:\javac app.java
Note: app.java uses or overrides a deprecated API. Recompile with "-
deprecation" for details.
1 warning
```

Esto no es de mucha ayuda. Sin embargo, utilizando la opción -deprecation, puede conocer el problema exacto:

```
C:\>javac app.java -deprecation
app.java:109: Note: The method java.awt.Dimension size() in class
java.awt.Component has been deprecated.
        x = (size().width - fontmetrics.stringWidth(text))/2;
                  ^
Note: app.java uses or overrides a deprecated API.  Please consult
the documentation for a better alternative.
1 warning
```

Como puede ver, el problema es que app.java utiliza el método size en la línea 109 y ese método es obsoleto. Si se reemplaza por el método de la nueva versión, getSize, se resuelve el problema.

Ejecución del código

El gran jefe se está poniendo impaciente. Ha escrito usted una nueva aplicación y la ha compilado sin errores a la primera (de lo que puede sentirse orgulloso), pero realmente no ha ocurrido nada que el gran jefe pueda ver. Es hora de ejecutar la nueva aplicación.

Las aplicaciones Java se ejecutan con el programa llamado java (en Windows, por ejemplo, es el archivo java.exe del directorio bin de Java). El programa java, llamado utilidad java, es lo que realmente ejecuta JVM. A continuación puede ver cómo usar la utilidad java:

```
java [opciones] clase [parámetro ...]
java [opciones] -jar archivo.jar [parámetro ...]
```

Estos son los parámetros de las líneas anteriores:

- opciones. Opciones de la línea de comandos, que se verán en otro apartado.

- clase. Nombre de la clase a la que se llama.
- archivo.jar. Nombre del archivo de archivo Java (JAR) al que se llama. Se usa sólo con –jar. Los archivos JAR se tratan en el capítulo 23.
- Parámetro. Un parámetro de la línea de comando que se pasa al método main.

Por ejemplo, para ejecutar la aplicación llamada app, que está en el archivo app.class, podría ejecutar el siguiente comando en la línea de comandos (observe que se omite la extensión".class" de app.class):

```
java app
```

El resultado aparecerá inmediatamente:

```
java app
¡Hola desde Java!
```

En la figura 1.4 puede ver cómo funciona en una ventana DOS bajo Windows.

Eso es todo. Ya ha escrito, compilado y ejecutado su primera aplicación. ¡Felicidades!

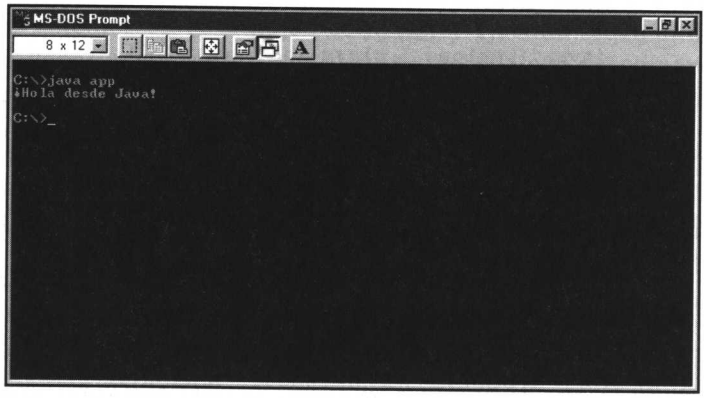


Figura 1.4. Ejecución de una aplicación en una ventana DOS.

Observe que si su aplicación no responde o si quiere pararla por alguna razón, puede pulsar Control-C. Si no funciona, inténtelo con la tecla Esc.

Además, en este libro, verá cómo crear aplicaciones ventana, y cuando ejecute una de estas aplicaciones con la utilidad java, obtendrá los resultados que aparecen en la figura 1.5.

Hay algo que hacer notar sobre la figura 1.5; la ventana consola (aquí una ventana DOS) está por debajo y espera que la aplicación termine antes de

continuar (es decir, antes de que el *prompt* de DOS aparezca en este caso). Si no quiere que una ventana consola esté asociada con su aplicación ventana, puede utilizar la utilidad javaw, como se indica:

```
javaw app
```

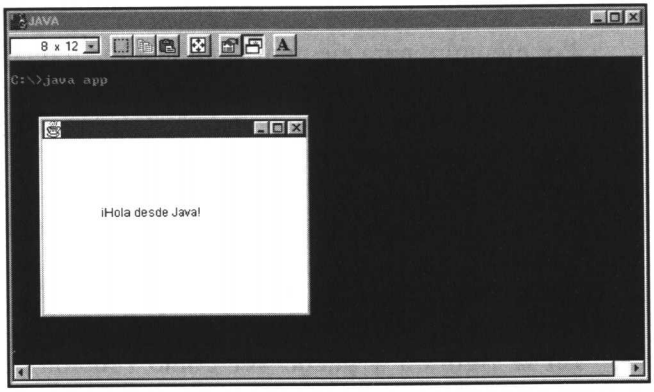


Figura 1.5. Ejecución de una aplicación ventana.

A continuación se indica cómo se utiliza javaw en general, igual que la utilidad java:

```
javaw [opciones] clase [parámetro...]
javaw [opciones] -jar archivo.jar [parámetro...]
```

Estos son los parámetros utilizados con javaw:

- opciones. Opciones de la línea de comandos, que trataremos próximamente.
- clase. Nombre de la clase a la que se llama.
- Archivo.jar. Nombre del archivo de archivo Java (JAR) al que se llama. Se usa sólo con –jar. Los archivos JAR se tratan en el capítulo 23.
- Parámetro. Un parámetro de la línea de comandos pasado al método main.

Cuando usted lanza una aplicación ventana de Java como esta, la ventana consola no espera a que la aplicación termine; si la está ejecutando en DOS, la aplicación aparece y el prompt DOS reaparece en la ventana DOS. Esto da un sentido más profesional a estas aplicaciones.

De hecho, hay otro lanzador en Java 2, el lanzador *oldjava*, que Sun incluyó para la compatibilidad hacia atrás. Este lanzador no soporta el *framework* de extensiones Java (ver el punto "¿Cuáles son las novedades de

Java 2?"). Proporciona compatibilidad hacia atrás cuando tiene una aplicación que utiliza el estilo de seguridad de Java1.1, que es incompatible con las técnicas de carga de clases de 1.2 (o quizás las clases que se están cargando han sido generadas o cambiadas de alguna forma que es incompatible con la estructura de clases de 1.2). En este libro estoy siguiendo Java 2, por lo que no se usará mucho el oldjava, pero si está migrando a Java 2 y necesita usar el mecanismo de carga de clases antiguo, es bueno saber que está ahí. También hay una utilidad oldjavaw.

Así es como puede ejecutar una aplicación, usando cualquier utilidad de Java: java, javaw u oldjava. Cuando lance una aplicación java2, el compilador *Just In Time* (JIT) compila los *bytecodes* y ejecuta la aplicación.



Truco: si no quiere utilizar el compilador JIT por alguna razón, hay dos formas para deshabilitarlo. Puede fijar la variable de entorno JAVA_COMPILER a NONE, usando, por ejemplo, el comando SET de Windows 95/98 o el panel de control del sistema en Windows NT. Además puede usar la opción –D de la línea de comandos para poner la palabra java.compiler a NONE, de la siguiente forma: java -Djava.compiler=NONE app.

Mientras estemos en el tema de compilar y ejecutar el código, hay otro detalle que deberíamos tratar: las opciones de la línea de comando que se pueden usar con los comandos javac y java. Les echaremos un vistazo en los dos apartados siguientes.

Ejecución de código: utilizar las opciones de la línea de comandos

"Bien," dice el programador, "tengo otro problema. He almacenado todos mis archivos con extensión ".class" en un directorio, pero no quiero estar cambiando a ese directorio para ejecutarlos". "Otro problema fácil de resolver," le dice. "Puede usar la opción –classpath de la utilidad java o establecer la variable de entorno CLASSPATH para que el compilador de Java busque las clases correctamente".



Nota: para más detalles sobre CLASSPATH, tema importante en la programación Java, ver el punto "Conocimientos básicos: búsqueda de clases Java con CLASSPATH".

Echaremos un vistazo al uso de las opciones de la línea de comandos en este punto. Utilizará estas opciones con las utilidades java, javaw y oldjava, de la siguiente forma (para ver información sobre estas utilidades, ver la sección "Ejecución del código," ya tratado en este capítulo):

```
java [opciones] clase [parámetro...]
java [opciones] -jar archivo.jar [parámetro...]
javaw [opciones] clase [parámetro...]
javaw [opciones] -jar archivo.jar [parámetro...]
oldjava [opciones] clase [parámetro...]
oldjavaw [opciones] -jar archivo.jar [parámetro...]
```

A continuación se citan las opciones de la línea de comandos que puede utilizar con estas herramientas (observe que las opciones no estándar, que quiere decir que en un futuro puede que no estén soportadas, empiezan con una X):

- -classpath ruta de acceso o –cp ruta de acceso. Especifica una lista de directorios, archivos con extensión".jar" o archivos con extensión".zip" para localizar archivos de clase. Se separan las entradas con punto y coma (';'). Observe que al especificar cualquiera de estas dos opciones se está borrando lo que indique la variable de entorno CLASSPATH. Utilizadas con java o javaw, -classpath o –cp sólo especifican la ruta de acceso para las clases de usuario. Usadas con oldjava u oldjavaw, especifican la ruta de acceso para las clases de usuario y las de arranque. Si no se especifican ni –classpath ni –cp y CLASSPATH no está, la ruta de acceso a las clases de usuario está limitada al directorio actual, que está referenciado con un punto ('.'). Ver el apartado "Conocimientos básicos: búsqueda de clases Java con CLASSPATH," más adelante en este capítulo para más información.
- -Dproperty=valor. Fija un valor de propiedad del sistema.
- -jar. Ejecuta un programa encapsulado en un archivo JAR. El primer parámetro es el nombre de un archivo JAR en vez de un nombre de clase de inicio. Cuando usa esta opción, el archivo JAR es la fuente de todas las clases de usuario y el resto de las rutas de acceso a otras clases de usuario se ignoran. Las herramientas oldjava y oldjavaw no soportan la opción –jar.
- -verbose o –verbose:clase. Visualiza información sobre cada clase que está cargada.
- -verbose:gc. Informa sobre cada evento de la colección *garbage*. Esta colección realiza la gestión de memoria en Java.

- -verbose:jni. Da información sobre el uso de métodos nativos (es decir, específicos de la plataforma) y otras actividades de la interfaz nativa de Java.
- -version. Visualiza información sobre la versión y se cierra.
- -? o –help. Visualiza información de ayuda y se cierra.
- -X. Visualiza información sobre las opciones no estándares.
- -Xbootclasspath:ruta de acceso de arranque. Especifica una lista de directorios separados por punto y coma (';'), archivos con extensión ".jar" o archivos con extensión ".zip" para localizar los archivos de la clase de arranque. Observe que estos serán utilizados en lugar de los archivos de las clases de arranque que Java incorpora.
- -Xdebug. Inicia con el debugger habilitado.
- -Xnoclassgc. Deshabilita la colección de clases *garbage*.
- -Xmsn. Indica el tamaño de memoria inicial que se quiere utilizar (este valor debe ser mayor que 1000). Para multiplicar el valor por 1000, añadir la letra 'k'. Para multiplicar el valor por un millón, añadir la letra 'm'. El valor por defecto es 1m.
- -Xmxn. Especifica el tamaño máximo de memoria (este valor debe ser mayor que 1000). Para multiplicar el valor por 1000, añadir la letra 'k'. Para multiplicar el valor por un millón, añadir la letra 'm'. El valor por defecto es 64m.
- Xrunhprof[:help][:<subopción>=<valor>,...]. Habilita el seguimiento de la CPU, la pila o del monitor. Esta opción va seguida normalmente por una lista de pares separados por coma (',') de la forma <subopción>=<valor>.
- -Xrs. Reduce el uso de las señales del sistema operativo.
- Xcheck:jni. Ejecuta tratamientos adicionales para las funciones de la interfaz nativa de Java.
- -Xfutrue. Ejecuta un seguimiento estricto del formato de archivo de clases.

Conocimientos básicos: comentar el código

La persona encargada de la corrección del programa entra y le mira con reproche. "¿Qué ocurre?, pregunta. "Su código", dice. "No puedo imaginar lo

que ahí está escrito". "Lo siento, olvidé comentarlo", dice. "Le pido que lo haga", dice él. "Arréglelo".

A veces, el código puede ser muy críptico y difícil de descifrar. Por ello, Java le permite comentar su código para explicar a todo el mundo que lo lea cómo funciona el programa y lo que hace. Como ejemplo, añadamos comentarios a la aplicación que ya hemos desarrollado en los puntos anteriores:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Java admite tres tipos de comentarios, dos de los cuales son de C++. Puede poner cualquier comentario de cualquier longitud con los caracteres "/*" y "*/":

```
/* Esta aplicación visualiza "¡Hola desde Java!" */
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

El compilador de Java ignorará todo el texto que esté entre "/*" y "*/". Puede dividir los comentarios que ponga entre estos indicadores en varias líneas:

```
/* Esta aplicación visualiza "¡Hola desde Java!"
        Creado por: G. Whiz, 1/1/00                    */
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

De hecho, en muchos entornos corporativos, se espera que se use una cabecera de comentario estándar, creado con "/*" y "*/" para todo código nuevo. Sería algo así como:

```
/*****************************************************
*  Esta aplicación visualiza "¡Hola desde Java!"                *
*                                                               *
*  Autor: G.Whiz                                                *
```

```
* Importaciones: Ninguna                                    *
* Parámetros: Argumentos de la línea de comandos            *
* Retorno: Ninguno                                          *
* Supuestos: Ninguno                                        *
* Fecha de creación: 1/1/00                                 *
* Última actualización: 1/1/01                              *
*************************************************/

public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Java también soporta una línea de comentario utilizando la doble barra, "//". El compilador de Java ignorará toda la línea después de la doble barra, por lo que puede crear líneas enteras de comentarios o añadir un comentario a una sola línea:

```
/* Esta aplicación visualiza "¡Hola desde Java!" */
    public class app          //Crear la clase app
    {
        //Crear main( ), punto de entrada a la aplicación.
        public static void main(String[] args)
        {
            // Visualizar el mensaje con
            System.out.println("¡Hola desde Java!");
        }
    }
```

Finalmente, Java también soporta un comentario que se inicia con "/**" y que termina con "*/". Este comentario está diseñado para usarse con la herramienta javadoc, que permite crear documentación casi automáticamente. Echaremos un vistazo a esto en el capítulo 21. A continuación hay un ejemplo para el uso de "/**" y "*/":

```
/** Esta aplicación visualiza "¡Hola desde Java"! */
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Comentar el código es de gran importancia en entornos de grupo donde se comparten los códigos de los archivos fuente. Además es muy cómodo si otra persona va a retomar el proyecto en el que ha estado trabajando usted.

Conocimientos básicos: importando paquetes y clases Java

"Hmm", dice el programador novato, "tengo un problema. El coordinador del equipo de diseño me dijo que usara la clase Date para visualizar la fecha actual en mi aplicación, pero Java parece no entender la clase Date, me da un error cada vez que intento usarlo". "Eso se debe a que la clase Date forma parte del paquete de utilidades de Java y tiene que importar ese paquete antes de que pueda utilizarlo". "¿Importarlo?", pregunta el programador.

Las clases que Sun ha creado están almacenadas en librerías de clases llamadas "paquetes". Para que una clase de un paquete esté disponible en su código, tiene que importar el paquete, lo que quiere decir que el compilador localizará ese paquete para esas clases. Además puede importar clases individuales que no son parte de un paquete. Por defecto, sólo las sentencias básicas de Java están disponibles en una aplicación, las del paquete java.lang. El compilador automáticamente importa el paquete java.lang, pero para usar el resto de las clases que vienen con Java tendrá que hacer la importación con la instrucción import. Aquí se muestra cómo utilizar esa sentencia:

```
import [paquete1[.paquete2...].](nombre de clase|*);
```

Observe que se pone un punto ('.') entre el paquete y los nombres de clase para separarlos. Los paquetes Java estándares están almacenados en un gran paquete llamado java, por lo que el paquete de utilidades se llama realmente java.util (hay otros grandes paquetes disponibles; por ejemplo, el paquete *swing* está almacenado en el paquete java). Puede referirse a la clase Date en java.util como java.util.Date. A continuación se indica cómo importar esa clase en un programa:

```
import java.util.Date;
public class app
{
```

Observe que si va a usar las sentencias import para importar clases en un programa, éstas deberían estar en la parte de arriba del código. Así, podemos utilizar la clase Date, como se indica a continuación (observe que estamos creando un objeto de la clase Date usado el operador Java new, de la que aprenderemos más en el capítulo 4):

```
import java.util.Date;

public class app
{
```

```
    public static void main(String[] args)
    {
  System.out.println("Hoy = " + new Date( ));
    }
}
```

Cuando ejecute esta aplicación, verá la fecha del día:

```
C:\>java app
Hoy = Lunes 2 Agosto 12:15:13 EDT 2000
```

Como puede ver estudiando la forma general de la sentencia previa import, hay también una técnica taquigráfica que carga todas las clases de un paquete; se puede usar un asterisco ('*') para utilizar todas las clases de un paquete específico. A continuación se indica cómo se podría hacer si quisiera importar todas las clases del paquete java.util de una vez:

```
import java.util.*;
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Hoy = " + new Date());
    }
}
```



Truco: la importación de paquetes y clases sólo indica al compilador dónde buscar el código que necesita, no incrementa el tamaño del código. Por esa razón, el archivo de *bytecodes* app.class será del mismo tamaño que si usara cualquiera de las siguientes sentencias: import java.util.Date, o import java.util.*.

Esto está bien si se quiere importar las clases proporcionadas por Sun, ya que Java sabe donde buscar las clases con las que fue instalado. Pero ¿qué pasa si quiere importar sus propias clases u otras proporcionadas por un tercero?

He aquí hay un ejemplo. Supongamos que tiene una clase llamada printer en un archivo llamado printer.java, y esa clase tiene un método, llamado print:

```
public class printer
{
  public void print( )
  {
      System.out.println("¡Hola desde Java!");
  }
    }
```

Podría querer utilizar el método print en otras clases, como en este caso, donde estamos creando un nuevo objeto de la clase printer usando el operador new y usando el método printer de ese objeto en una aplicación llamada app:

```
public class app
{
    public static void main(String[] args)
    {
      (new printer( )).print( );
  }
    }
```

Para ello, puede importar la clase printer de esta forma (observe que puede además poner el código de la clase printer en el mismo archivo que la clase app, en cuyo caso no tendría que importar la clase printer):

```
import printer:
public class app
{
    public static void main(String[] args)
    {
        (new printer()).print();
    }
}
```

Esto funciona como debería. Felicidades, acaba de importar una clase en su programa.

Esta técnica es buena si printer.class está en el mismo directorio en el que se está compilando esta aplicación, ya que el compilador Java buscará el directorio actual por defecto. Sin embargo, supongamos que quiere almacenar todas sus clases en un directorio llamado c:\clases. ¿Cómo encontrará el compilador printer.class allí? Para contestar a esa pregunta, veamos el siguiente punto CLASSPATH.

Conocimientos básicos: buscar clases Java con CLASSPATH

"Ese jorobado Johnson", dice el programador novato. "Me dio un archivo de clases nuevo, johnson.class, para trabajar con él, y se suponía que resolvería mis problemas con esa hoja de cálculo. Pero Java dice que no puede encontrar johnson.class". "¿Dónde tiene ese archivo?", le pregunta. "En un directorio especial que creé para ello", dice el PN, "llamado jorobadojohnson". "Ese es el problema", le dice. "Tiene que incluir el directorio jorobadojohnson en su ruta de acceso a clases".

Por defecto, Java podrá encontrar sus clases de arranque (con las que viene), extensiones de clases (esas que usan el *framework* de extensiones Java; ver en este mismo capítulo) y clases en el directorio actual (es decir, donde está compilando su programa). Las clases pueden almacenarse en

archivos con extensión ".class", ".jar" y ".zip". Java puede localizar todo este tipo de archivos.

Pero ¿qué ocurre si quiere hacer una búsqueda de clases en otro directorio o en un archivo con extensión ".jar" suministradas por un tercero? Puede hacerse con la variable de entorno CLASSPATH, ya que Java usa esta variable para determinar dónde quiera buscar las clases.

Veamos un ejemplo que introduje en el punto anterior. Digamos que tiene una clase llamada printer en un archivo llamado printer.java, y esa clase tiene un método, llamado print:

```
public class printer
{
  public void print ( )
  {
      System.out.println("¡Hola desde Java!");
  }
}
```

Ahora supongamos, como en el punto anterior, que quiere usar el método print de otras clases, como en este caso, donde creamos un nuevo objeto de la clase printer usando el operador new y usando el método print de ese objeto en una aplicación llamada app:

```
import printer;

public class app
{
    public static void main(String[] args)
    {
       (new printer( )).print( );
  }
     }
```

Esto funciona si printer.class está en el mismo directorio en el que se está compilando esta aplicación, porque el compilador de Java buscará el directorio actual por defecto. Pero supongamos que quiera almacenar todas sus clases en un directorio llamado c:\clases. ¿Cómo buscará el compilador Java printer.class ahí?

Para que el compilador Java busque en c:\clases, puede fijar la variable de entorno CLASSPATH para que incluya ese directorio. Por defecto, no hay ninguna ruta de acceso ni directorios en CLASSPATH, pero puede añadir a CLASSPATH una lista con separaciones por punto y coma (';'), como se indica a continuación para Windows (observe que es importante no dejar espacios entre los signos de igual):

```
SET CLASSPATH=c:\clases;c:\newclasses
```

En Windows NT, puede seguir estos pasos para fijar la variable de entorno CLASSPATH:

1. Abrir el menú Inicio y abrir el Panel de Control de la opción Configuración. Hacer doble clic sobre Panel de Control para abrirlo.
2. En el cuadro de diálogo Propiedades del Sistema, hacer clic sobre la pestaña Entorno.
3. Hacer clic sobre la variable CLASSPATH, haciendo que se vea el valor actual de CLASSPATH al final del cuadro de diálogo.
4. Añadir la ruta de acceso que quiera a CLASSPATH y hacer clic sobre OK para cerrar el cuadro de diálogo de Propiedades del Sistema.

También puede saber el valor actual de CLASSPATH usando el comando SET:

```
C:\>SET
TMP=C:\WINDOWS\TEMP
PROMPT=$p$g
winbootdir=C:\WINDOWS
COMSPEC=C:\WINDOWS\COMMAND.COM
PATH=C:\WINDOWS;C:\JDK1.2.2\BIN
windir=C:\WINDOWS
CLASSPATH=C:\CLASES;C:\NEWCLASSES
```

Ahora el compilador de Java (y otras herramientas Java como la utilidad java) tendrá suficiente información para buscar en c:\clases y c:\newclasses automáticamente. Esto significa que este código funcionará si printer.class está en c:\clases, ya que ese directorio está en CLASSPATH:

```
import printer;

public class app
{
    public static void main(String[] args)
    {
        (new printer()).print();
    }
}
```

Puede añadir una nueva ruta de acceso en CLASSPATH a la que ya existe, como sigue:

```
SET CLASSPATH=c:\clases;c:\newclasses;%CLASSPATH%
```

Observe que también puede buscar archivos con extensión ".jar" y ".zip" para las clases:

```
SET CLASSPATH=server.jar;classes.zip;%CLASSPATH%
```

En los orígenes, CLASSPATH produjo dolor de cabeza a los programadores de Java ya que ninguna clase era considerada clase de arranque, lo que significaba que había que fijar y entender CLASSPATH antes de que se pudiera usar Java. Esto se ha arreglado con el concepto de clases de arranque, que son clases que vienen con Java (y son localizadas automáticamente). Sin embargo, si quiere usar paquetes no estándares y almacenar sus propias clases en otros directorios, es importante saber cómo se fija CLASSPATH.

Crear *applets*

El gran jefe se está impacientando. "¿Qué es todo esto de las aplicaciones que visualizan "¡Hola desde Java!" en la ventana consola? Queremos usar Java para crear applets que se puedan utilizar en los navegadores Web". "De acuerdo", le dice, "deme un minuto".

En los capítulos que siguen, echaremos un vistazo a la sintaxis de Java, que sería un duro camino si lo primero que se quisiera hacer fuera escribir *applets* en primer lugar. Lo que es más, sería intolerable si no iniciáramos un libro de un lenguaje tan visual como Java con al menos una *applet*. En este punto, cubriré el proceso de crear una *applet* Java. El saber crear una *applet* básica le ayudará a probar, visualmente, la sintaxis de los siguientes capítulos. Las *applets* se introducirán formalmente en el capítulo 6; por lo tanto, considere esto como un aperitivo.

Las *applets* estándar están construidas en la clase Applet, que está en el paquete java.applet. Sin embargo, comenzaremos importando esa clase en un nuevo archivo de código fuente Java, que llamaremos applet.java:

```
import java.applet.Applet;
```

La clase java.applet.Applet es la clase que forma la base para las *applets* estándar, y puede derivar sus propias clases de applets de esta clase usando la palabra clave extends:

```
import java.applet.Applet;

public class applet extends Applet
{


}
```

Es hora de añadir código a esta nueva *applet*. Las *applets* no tienen un método main como las aplicaciones; de hecho, esa es la diferencia principal

entre applets y aplicaciones. Por lo tanto, ¿cómo se puede visualizar texto directamente en una *applet*?

El trazado actual de una *applet* está contenido en su método *paint*, que la máquina virtual de Java llama cuando es hora de visualizar la *applet*. La clase java.applet. Applet tiene su propio método *paint*, pero podemos sobrescribir el método definiendo su propio método *paint*, como sigue (ver capítulos 4 y 5 para más detalles de la sobrescritura):

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
  public void paint(Graphics g)
  {


  }
    }
```

Este método *paint* es realmente una parte de Java Abstract Windows Toolkit (AWT), que verá con gran detalle en este libro, por lo que hemos importado las clases AWT con la instrucción import.java.awt.*. Verá cómo el siguiente método paint se pasa a un objeto Java de la clase Graphics (este objeto se llama g en el código). Puede usar el método drawString de este objeto para dibujar el texto.

En este caso, dibujaremos el texto "¡Hola desde Java!" en las coordenadas (60,100) en la *applet*; las coordenadas se miden en pixels desde la parte superior izquierda de la *applet*, por lo que esta posición está a 60 pixels del borde izquierdo de la *applet* y a 100 pixels del el superior. Este es el código:

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public void paint(Graphics g)
    {
      g.drawString("¡Hola desde Java!", 60, 100);
  }
     }
```

Eso es todo, ahora puede compilar applet.java para obtener applet.class. Hay un paso más: crear una página Web para visualizar en ella la *applet*. Veremos esto a continuación.

Ejecutar *applets*

"De acuerdo", dice el gran jefe, "ha creado una *applet*. ¿Por qué no lo veo en una página Web?" "Veámoslo," dice. "Creo...".

Para visualizar una *applet*, puede usar una página Web con la etiqueta <APPLET>de HTML (*Hypertext Markup Language*). De hecho, se puede almacenar el HTML necesario en un archivo de código fuente de *applets*, como podrá ver en el capítulo 6; además lo aprenderá todo sobre la etiqueta <APPLET>en ese capítulo. Por ahora, aquí hay una página Web, applet.html, que visualizará la *applet* desarrollada en el punto anterior:

```
<HTML>
<HEAD>
<TITLE>APPLET</TITLE>
<HEAD>
<BODY>
<HR>
<CENTER>
<APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
</CENTER>
<HR>
</BODY>
</HTML>
```

Puede abrir esta *applet* en una página Web con un navegador Web, como se ve en la figura 1.6, donde la *applet* se ha abierto con Microsoft Internet Explorer.

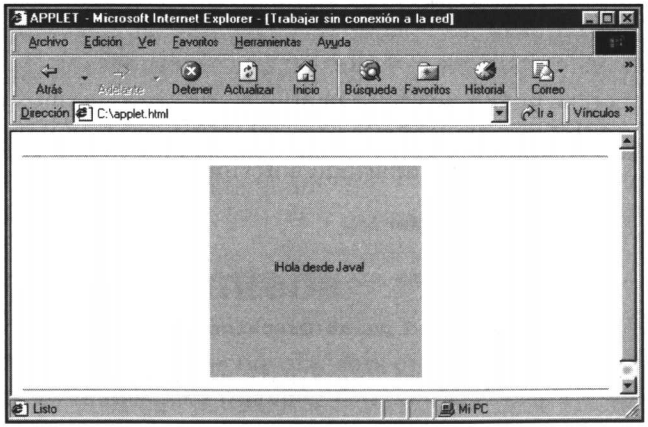


Figura 1.6. Una *applet* funcionando con Internet Explorer.

Además puede usar el visualizador de applets de Sun, que viene con Java, para abrir applet.html, de la siguiente forma:

```
C:\>appletviewer applet.html
```

La figura 1.7 muestra la *applet* con el visualizador de applets de Sun.

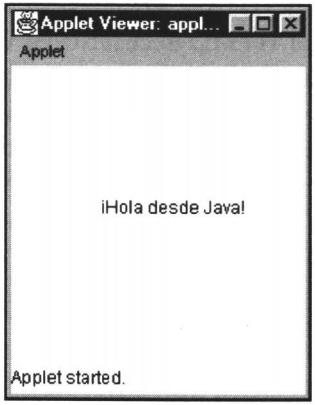


Figura 1.7. Una *applet* funcionando en el visualizador de applets de Sun.

Crear aplicaciones ventana

El gran jefe está impresionado con su nueva *applet* y pregunta, "¿Puede también hacer que una aplicación visualice ventanas?" "Por supuesto", le dice. "Vayamos a ello".

Lo aprenderá todo sobre la creación de aplicaciones ventana en el capítulo 6, pero tomaremos un primer contacto aquí. Crear una aplicación ventana es parecido a la creación de una *applet*, con la excepción de que tiene que tener un método *main*, y que usted es el responsable de crear la ventana. Con el fin crear la ventana para la aplicación, derivaremos una nueva clase de la clase AWT *Frame* y añadiremos el mismo código del método *paint* que se utilizó en la *applet* del apartado anterior:

```
import java.awt.*;

class AppFrame extends Frame
{
  public void paint(Graphics g)
  {
      g.drawString("¡Hola desde Java!", 60, 100);
  }
    }
```

Ahora crearemos la clase aplicación, que llamaremos app. Esta es la clase que tendrá un método *main*, y en ese método, usaremos el operador new para crear un nuevo objeto de la clase AppFrame al que se le dará un tamaño en pixels y se mostrará en la pantalla:

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

class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}

public class app
{
    public static void main(String [] args)
    {
      AppFrame f = new AppFrame( );

      f.setSize(200, 200);

      f.addWindowListener(new WindowAdapter( ) {public void
            windowClosing(WindowEvent e)  {Sytem.exit(0);}});

      f.show( );
   }
    }
```



Truco: incluir aquí la línea de códigos que tiene que hacerse con el método addWindowListener significa que cuando la ventana de la aplicación se cierra, la aplicación terminará. Aprenderá más sobre cómo compactar y hacer más potente la línea, que usa clases internas y clases adaptadoras, más adelante.

Ahora que la nueva aplicación ventana está preparada, ¿cómo se ejecuta? Echemos un vistazo al siguiente punto.

Ejecutar aplicaciones ventana

Como con las aplicaciones consola, para ejecutar una aplicación ventana, puede usar las utilidades java o javaw:

```
java app
javaw app
```

La utilidad java lanza la aplicación y hace que la ventana consola espere hasta que la aplicación sea desechada, mientras que la utilidad javaw lanza la aplicación y no espera a que la aplicación sea desechada. La ejecución de la aplicación aparece en la figura 1.8.

Esto es todo; ahora ejecutaremos aplicaciones ventana Java.



Figura 1.8. Una aplicación ventana.

Diseño de programas Java

El diseño de la cabecera del programa ya está hecho, y su nueva oficina funciona. Pero cuando se sienta usted allí, mirando fijamente la esquina de la ventana y golpeando su nueva mesa, se pregunta si es capaz de conservar su nueva posición.

El diseño de los programas en Java no es tarea fácil. Un buen diseño de programación involucra muchos aspectos, y echaremos un vistazo a algunos de ellos en este capítulo, antes de empezar a excavar en la sintaxis de Java.

De hecho, uno de los aspectos más importantes de la creación de una nueva aplicación es diseñarla. Si no se selecciona bien puede que sea necesario hacer muchas revisiones del producto. Muchos libros se dedican al diseño de programas. Microsoft, que debería saber algo de ello, divide el proceso en cuatro áreas:

- Rendimiento. Responsabilidad y optimización global de la velocidad y uso de recursos.
- Mantenimiento. Capacidad de la aplicación para ser mantenida fácilmente.
- Extensibilidad. Capacidad de la aplicación para ser extendida de formas bien definidas.

- Disponibilidad. Robustez de la implementación de la aplicación y disponibilidad para su uso.

Veamos rápidamente estas cuatro áreas.

Rendimiento

El rendimiento es un tema de diseño que es duro de argumentar. Si los usuarios no consiguen lo que quieren con su aplicación, esto se convierte claramente un problema. En general, el rendimiento depende de las necesidades de los usuarios. Para algunas personas, la velocidad es esencial; para otros, la robustez o el uso eficiente de los recursos es lo que están buscando. Globalmente, el rendimiento de una aplicación es una indicación de lo bien que responde a las necesidades de los usuarios. He aquí algunos aspectos generales de rendimiento que debería considerar cuando escriba programas Java:

- eficiencia del algoritmo
- velocidad de CPU
- diseño y normalización eficiente de base de datos
- limitación de accesos externos
- velocidad de la red
- temas de seguridad
- uso de recursos
- velocidad de acceso a la Web

A lo largo del libro trataremos más temas específicos de rendimiento.

Mantenimiento

El mantenimiento es la medida de lo fácilmente que puede adaptarse su aplicación a necesidades futuras. Este asunto se deriva de buenas prácticas de programación, de las que hablaré a lo largo del libro. Buena parte de esto es de sentido común, simplemente tener en mente las necesidades de codificación futura al escribir el código. Algunos puntos de la "programación óptima" son los siguientes:

- Evitar el uso de bucles y condicionales anidados.
- Evitar el paso de variables globales a procedimientos

- Ser modular cuando se escribe el código.
- Dividir el código en paquetes.
- Documentar los cambios de programa.
- Dar a cada procedimiento un único propósito.
- Asegurarse de que la aplicación puede extenderse sin problemas a más tareas y mayor número de usuarios.
- Planificar la reutilización de código.
- Programar de forma defensiva.
- Uso de procedimientos para el acceso a datos sensibles.
- Uso de comentarios.
- Uso de nombres de variables consistentes.
- Uso de constantes en lugar de números "mágicos".

Extensibilidad

La extensibilidad es la capacidad de la aplicación para extenderse de una forma bien definida y relativamente fácil. Generalmente, supone una preocupación en las aplicaciones grandes y, con frecuencia, involucra a toda una interfaz especialmente diseñada para la extensión de módulos. De hecho, Java, en sí mismo, está diseñado para ser extendido, usando el *framework* de extensiones Java.

Disponibilidad

La disponibilidad es medir del tiempo que la aplicación puede utilizarse, en comparación con el tiempo que los usuarios quieren utilizarla. Esto lo incluye todo, desde que no se quede congelada cuando se ejecuta una tarea larga (al menos, dar al usuario el estado de la operación), hasta trabajar con técnicas y métodos que no se cuelguen, hacer backups de datos críticos y planificar el uso alternativo de recursos, si es posible, cuando el acceso al recurso deseado esté bloqueado.

Globalmente el proceso de diseño es de los que más tiempo requiere. De hecho, todo el ciclo de desarrollo es tema de muchos estudios, puede resultar sorprendente saber que algunos de ellos consideran que el diseño es, al menos, el quince por ciento del proyecto total cuando se prueba un campo y se añaden planificación, diseño y pruebas de interfaz de usuario.

No entraremos en más detalle sobre el ciclo de desarrollo de software, pero los programadores no deberían cambiar los pasos críticos del diseño en proyectos serios, ya que pueden dar más problemas en una ejecución larga que el tiempo que ahorran en una ejecución corta.

Distribución del programa Java

"Bien", dice el programador novato, "he terminado mi programa Java, y estoy preparado para venderlo". "Ah, ¿sí?" pregunta. "Revisaremos primero los temas de licencia".

Para que los usuarios ejecuten sus programas, necesitarán tener un entorno de ejecución Java en sus sistemas. El JDK de Java 2 contiene un entorno de ejecución, por lo que los usuarios podrían usar su programa si lo tuvieran instalado. Sin embargo, observe que la mayoría de los usuarios no tendrán todo el JDK Java 2 instalado, por lo que la mejor selección para los usuarios será el entorno de ejecución de Java 2 (JRE). Aquí es donde la distribución de JRE en lugar de JDK ocupa un lugar destacado:

- El entorno de ejecución de Java es redistribuible, y el JDK de Java 2 no, lo que viene a decir que la licencia de JRE le permite empaquetarlo con su software. Mediante la distribución de JRE de su aplicación, puede asegurarse de que sus usuarios tendrán la versión correcta del entorno de ejecución de su software.
- El JRE es más pequeño que el JDK. El JRE contiene todo lo que los usuarios necesitan para ejecutar su software, pero no incluye las herramientas de desarrollo y las aplicaciones que forman parte del JDK. Dado que el JRE es pequeño, es más fácil empaquetarlo con su software en vez de que los usuarios lo descarguen del sitio Web de software Java.
- En Windows, el instalador JRE instala automáticamente java y javaw en la ruta de acceso del sistema operativo, lo que significa que no tiene que preocuparse de encontrar los lanzadores para iniciar la aplicación (es decir, no tiene que dar instrucciones a los usuarios para que establezcan las rutas de acceso en sus sistemas).

Puede encontrar más información sobre el entorno de ejecución de Java 2 en http://java.sun.com/products/jdk/1.2/runtime.html.

2 Variables, arrays y cadenas

Este capítulo inicia nuestra discusión sobre la sintaxis de Java, y se verá gran parte de ella. Aquí, vamos a tratar sobre cómo se almacenan y recuperan, en Java, los datos de variables, *arrays* y cadenas. El trabajo con datos es parte fundamental de cualquier programa, y la información contenida en este capítulo es esencial para cualquier programador de Java. Aunque ya programe en Java, eche un vistazo al material de este capítulo, porque hay mucho que ver.

Variables

Las variables pueden ser de diferentes tipos y actúan como gestores de memoria de los datos. Los diferentes tipos tienen que ver con el formato de los datos que se almacenan en ellas, así como con la memoria que es necesaria para gestionar ese dato. Por ejemplo, la variable de tipo entero, *int*, es de 4 *bytes* (o 32 bits) y se utiliza para almacenar valores enteros. Esto hace que un dato de tipo *int* pueda tomar un rango de valores que va desde –2.147.483.648 hasta 2.147.483.647. En Java, hay bastantes tipos de variables, por ejemplo, enteros, coma flotante y caracteres, que veremos en este capítulo.

Antes de usar una variable en Java, debe declararla, especificando su tipo. A continuación se indica cómo se declaran las variables en Java:

```
tipo nombre [ = valor][, nombre [ = valor]...];
```

Ahora, podemos ver cómo se declara una variable de tipo *int*, es decir que contiene un entero (la variable se llama *days*):

```
public class app
{
    public static void main(String[] args)
    {
  int days;


    }
}
```

Este código aloca 32 bits de memoria y etiqueta esa ubicación, de forma que el compilador entiende la variable *days* y en el código puede haber referencias a ese nombre. A continuación, se indica cómo se almacena un valor numérico de 365 en *days*, utilizando el operador de asignación de Java (=):

```
public class app
{
    public static void main(String[] args)
    {
        int days;
days = 365;



    }
}
```

En este caso, el valor 365 es una constante entera, es decir, un valor constante que se sitúa directamente en el código. A lo largo de este capítulo, veremos los tipos de constantes que Java permite. Para comprobar que *days* contiene el valor 365, lo visualizamos por pantalla:

```
public class app
{
    public static void main(String[] args)
    {
        int days;

        days = 365;

        System.out.println("Número de días = " + days);
    }
}
```

Este es el resultado del código:

```
C:\>java app
Número de días = 365
```

Como puede ver, hemos creado una variable en la que se han almacenado datos y se ha obtenido ese dato para visualizarlo en pantalla. Así funciona.

Hay un convenio que permite inicializar una variable en el momento de declararla. A continuación, se puede ver como *days* se declara e inicializa con el valor 365 en un único paso:

```
public class app
{
    public static void main(String[] args)
    {
  int days = 365;

  System.out.println("Número de días = " + days);
    }
}
```

El tipo *int* es una de las clases de variables que se pueden usar. Estas son todas las posibilidades:

- Enteros: Estos tipos son *byte, short, int* y *long*, que guardan el signo y el valor.
- Números en coma flotante: Estos tipos son *float* y *double*, que almacenan números en coma flotante con signo.
- Caracteres: Este es el tipo *char*, que guarda las representaciones de los caracteres, tanto letras como números.
- Booleano: Este tipo está diseñado para guardar sólo dos valores: verdadero (*true*) o falso (*false*).

Veremos todos estos tipos con más detalle más adelante, incluyendo el rango de valores que puede tomar cada uno. Todos ellos forman lo que en Java se conoce como tipos de datos sencillos. Cada uno de estos tipos representa un valor de datos sencillos, no uno compuesto (en oposición a un *array*, que también se tratará en este capítulo). Es posible almacenar un dato en una variable de cualquier tipo y ese dato debe estar dentro del rango permitido para ese tipo.

Tipos de datos

Java pone mucho énfasis en sus tipos de datos. Es un lenguaje que insiste en que las variables sencillas que se declaran y se utilizan deben corresponderse con la lista establecida.

Todas las variables sencillas deben tener un tipo (y de hecho, toda expresión, toda combinación de términos que Java puede evaluar para obtener un resultado, también tiene un tipo). Además, Java es muy particular para mantener la integridad de esos tipos, especialmente si intenta asignar un valor de un tipo a una variable de otro tipo. Java es más insistente en cuanto al tipo de datos que un lenguaje como C++; en C++, por ejemplo, se puede asignar un número en coma flotante a un entero y C++ gestionará la conversión de tipos, pero eso no se puede hacer en Java. Sin embargo, en Java, puede haber conversión entre ciertos tipos de datos, por ejemplo, entre tipos de enteros, como veremos en este capítulo.



Truco: cuando trabaje con variables, puede que el compilador de Java le dé muchos errores y avisos sobre los tipos de datos que hagan que no los utilice; hay que tener en cuenta que Java es tan particular en este asunto y que no se ha hecho más fácil para prevenir los errores del código.

Hasta aquí, hemos revisado lo que Java hace con los tipos de datos sencillos y variables; ya es hora de echar un vistazo al almacenamiento de datos, lo que se denomina *arrays*.

Arrays

Los tipos sencillos son buenos para almacenar datos simples, pero con frecuencia, los datos son más complejos. Supongamos, por ejemplo, que quiere crear un nuevo banco, el Banco de Programación Java, y necesita obtener la cantidad de dinero de cada cuenta, utilizando como índice el número de cuenta. En este caso, es mejor trabajar con datos compuestos, que son proporcionados por los *arrays*.

Utilizando un *array*, podrá agrupar tipos de datos sencillos en estructuras más complejas y hacer referencia a esa nueva estructura por su nombre. Lo que es más importante: mediante un índice numérico, podrá hacer referencia a los datos individuales almacenados en el *array*. Eso es importante, porque los ordenadores se destacan por ejecutar millones de operaciones muy rápidamente, por lo tanto si se puede hacer referencia a los datos con un índice numérico, se puede trabajar muy rápido con un conjunto de datos, simplemente incrementando el índice del *array* para así, acceder a todos sus elementos.

Aquí tenemos un ejemplo. En este caso, empezaré con 100 nuevas cuentas en el Banco de Programación de Java y cada una tendrá su propia entrada en

un *array* llamado accounts[]. Los corchetes indican que es un *array*, y entre ellos se sitúa el índice del elemento del *array* al que se quiere acceder. A continuación se puede ver cómo he creado el *array* accounts[], haciendo que cada elemento sea de tipo doble para obtener mucha precisión. Primero, declaro el *array* y luego lo creo con el operador *new*, que es el que Java utiliza para ubicar memoria:

```
public class app
{
    public static void main(String[] args)
    {
   double accounts[ ];

   accounts = new double[100];
```

Ahora que he creado un *array* con 100 elementos, puedo hacer referencia a ellos numéricamente, como se indica a continuación (observe que estoy almacenando $43.95 en la cuenta 3 y visualizando esa cantidad):

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[];

        accounts = new double[100];

        accounts[3] = 43.95;

        System.out.println("La cuenta 3 tiene $" = + accounts[3]);
    }
}
```

Este es el resultado del programa:

```
C:\>java app
La cuenta 3 tiene $43.95
```

Como puede ver, ahora puede referirse a los elementos del *array* utilizando un índice numérico, que los organiza fácilmente. En Java, el elemento más bajo declarado de esta forma es el 0, por lo que la sentencia accounts = new double[100] crea un *array* cuyo primer elemento es accounts[0] y el último es accounts[99].

Se pueden combinar los pasos de declaración y creación en uno solo:

```
public class app
{
    public static void main(String[] args)
    {
   double accounts[ ] = new double[100];
```

```
        accounts[3] = 43.95;

        System.out.println("La cuenta 3 tiene $" + accounts[3]);
    }
}
```

Además se puede inicializar un *array* con valores cuando se declara, encerrando entre llaves la lista de valores. Por ejemplo, este código crea cuatro cuentas y guarda el valor 43.95 en accounts[3]:

```
public class app
{
    public static void main(String[] args)
    {
  double accounts[ ] = {0, 0, 0, 43.95};
        accounts[3] = 43.95;

       System.out.println("La cuenta 3 tiene $" + accounts[3]);
    }
}
```

Esto hará que algunos clientes del Banco de Programación de Java no estén contentos, por supuesto; además de una cuenta corriente quieren una cuenta de ahorro. ¿Cómo gestionaremos esto y mantendremos todo indexado por número de cuenta?

El *array* accounts[] es un *array* unidimensional, también llamado vector, es decir, es una lista de números que pueden ser indexados por otro número. Sin embargo, en Java, los *arrays* pueden tener muchas dimensiones, lo que significa que se pueden tener muchos índices. En el siguiente ejemplo, extenderé accounts[] a un *array* de dos dimensiones, accounts[][], para gestionar la cuenta de ahorro y la cuenta corriente. El primer índice de accounts[][] será 0 para la cuenta de ahorro y 1 para la cuenta corriente, y el segundo índice será el número de la cuenta como antes. Así sería el código:

```
public class app
{
    public static void main(String[] args)
    {
  double accounts[ ][ ] = new double[2][100];

  accounts[0][3] = 43.95;
  accounts[1][3] = 2385489382.06;

  System.out.println("La cuenta de ahorro 3 tiene $" + account[0][3];
  System.out.println("La cuenta corriente 3 tiene $" + accounts[1][3];
    }
}
```

Ahora que accounts[][] es un *array* de dos dimensiones, la referencia a cada elemento se hace utilizando dos índices, por ejemplo, el saldo de la

cuenta de ahorro 3 es accounts[0][3] y el saldo de la cuenta corriente es accounts[1][3]. Estos son los resultados al ejecutar la aplicación:

```
C:\>java app
La cuenta de ahorro 3 tiene $43.95
La cuenta corriente 3 tiene $2.38548938206E9
```

Observe que he dado a la cuenta corriente 3 un saldo de $2,385,489,382.06 (ilusión) y que Java ha visualizado 2.38548938206 x 10^9.



Truco: a lo largo de este capítulo, verá muchos *arrays*, pero debería saber que Java 2 soporta muchas más estructuras de datos complejos además de los *arrays*. Estas estructuras de datos son inherentes al lenguaje. Java 2 soporta *hashes* y *maps* así como otros tipos de estructuras de datos, que verá cuando revisemos las nuevas colecciones de clases en el capítulo 21.

Cadenas

Quizás se haya dado cuenta de que, en los ejemplos anteriores, he usado el operador + para crear el texto que se va a visualizar:

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[][] = new double[2][100];

        accounts[0][3] = 43.95;
        accounts[1][3] = 2385489382.06;

     System.out.println("La cuenta de ahorro 3 tiene $" + accounts[0][3];
      System.out.println("La cuenta corriente 3 tine $" + accounts[1][3];
      }
}
```

Esto se debe a que las propias clases de Java (la clase *String*), soportan las cadenas de texto y se puede considerar la clase *String* como la definición de un nuevo tipo de datos. Por ejemplo, a continuación se puede ver cómo se crea una cadena llamada greeting que guarda el texto "¡Hola desde Java!":

```
public class app
{
    public static void main(String[] args)
```

```
    {
String greeting = "¡Hola desde Java!";
                .
                .
                .
```

Ahora, se puede tratar esta cadena de igual forma que los otros tipos de variables, incluyendo su visualización:

```
public class app
{
    public static void main(String[] args)
    {
        String greeting = "¡Hola desde Java!";

        System.out.println(greeting);
    }
}
```

Este es el resultado de la aplicación:

```
C:\>java app
¡Hola desde Java!
```

Aunque las cadenas no son uno de los tipos sencillos de Java, tienen un espacio en este capítulo, ya que la mayoría de los programadores los tratan como cualquier otro tipo de datos. De hecho, muchos programadores argumentarían que las cadenas deberían ser un tipo de datos sencillos en Java, como lo son en otros lenguajes. La razón es que no tienen nada que ver con la línea de Java, que extiende el lenguaje C. C no tiene un tipo sencillo de cadenas; en C, se gestionan las cadenas como *arrays* unidimensionales de caracteres, lo que es bastante embarazoso. Una de las cosas sobre C++ que hacía felices a los programadores era que la mayoría de las implementaciones incluían una clase *String*, que se podía usar igual que cualquier otro tipo de datos. Java continúa con este uso, implementando las cadenas como una clase, no como un tipo de dato intrínseco, pero la gestión de cadenas es tan fundamental para la programación, que tiene sentido revisar las cadenas en este capítulo.

Hay dos clases *string* en Java: *String* y *StringBuffer*. Se utiliza la clase *String* para crear cadenas de texto que no se pueden modificar. Como puede ver en el código anterior, puede utilizar cadenas como si fuera un tipo sencillo de Java. Echaremos un vistazo al uso de cadenas en este capítulo y en el siguiente (uso de operadores + y -). Revisaremos también el uso de operadores en las cadenas.

Por ahora, es suficiente. Comenzaremos a crear y a usar variables, *arrays* y cadenas.

¿De qué tipo de datos disponemos?

"Veamos", dice el gran jefe (GJ), "¿cómo escribiría un programa Java para gestionar la deuda de la empresa?" "¿Estamos en deuda?" pregunta. "Sólo un poco", responde GJ. "¿Cuánto?" le pregunta. "Aproximadamente $2.848.238.493.902,77", dice GJ. "Hmm", dice. "Creo que hay que usar números en coma flotante".

¿Qué tipo de datos sencillos puede usar para crear variables en Java? Los encontrará en la tabla 2.1.

Los tipos de datos sencillos pueden clasificarse por categorías, como se hizo al principio de este capítulo:

- Enteros: estos tipos son *byte, short, int* y *long*, que guardan el signo y el valor.

Tabla 2.1. Tipos de variables.

Tipo de variable	Almacenamiento en bytes	Rango de valores
boolean	2	Verdadero, Falso
byte	1	-128 a 127
char	2	N/A
double	8	-1.79769313486232E308 a -94065645841247E-324 para valores negativos y 4.94065645841247E-324 a 1.79769313486232E308 para valores positivos
float	4	-3.402823E38 a –1.401298E-45 para valores negativos y 1.401298E-45 a 3.402823E38 para valores positivos
int	4	-2,147,483,648 a 2,147,483,647
long	8	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
short	2	-32,768 a 32,767

- Números en coma flotante: estos tipos son *float* y *double*, que almacenan números en coma flotante con signo.

- Caracteres: este es el tipo *char*, que guarda las representaciones de los caracteres, tanto letras como números.
- Booleano: este tipo está diseñado para guardar sólo dos valores: verdadero (*true*) o falso (*false*).

Hasta aquí, una visión general de los tipos de datos sencillos que están disponibles. Veamos los siguientes puntos para ver cómo funciona cada uno de ellos.

Creación de literales enteros

El programador novato aparece y dice, "¿cómo asigno un valor hexadecimal, base 16, a una variable en Java?"

"Tiene que usar un literal hexadecimal, que empieza con los caracteres 0x o 0X", le responde.

Un literal es un valor constante que se puede usar directamente en el código Java, y hay una serie de reglas para controlarlo. En este capítulo, ya he usado literales enteros, que son los que uilizan la mayoría de los programadores. Este es el ejemplo anterior:

```
public class app
{
    public static void main(String[] args)
    {
  int days = 365;

  System.out.println("Número de días = " + days);
    }
}
```

Aquí, estoy asignando un literal entero con un valor de 365 a la variable *days*. Por defecto, los literales enteros son del tipo *int*. Sin embargo, si se les asigna a otros tipos de enteros, como *short*, Java convierte el tipo de literal automáticamente. Por otro lado, los valores *long* pueden tener más dígitos que los *int*, por lo que Java proporciona una forma explícita de crear constantes *long*: se añade una 'L' al final del literal. Este es el ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
        long value;
        value = 1234567890123456789L;

        System.out.println("El valor = " + value);
```

```
    }
}
```

Este es el resultado del código:

```
C:\>java app
El valor = 1234567890123456789
```

Además puede crear literales en formato octal empezando con un cero y en formato hexadecimal empezando con 0x o 0X. Estos son algunos ejemplos:

```
public class app
{
    public static void main(String[] args)
    {
        int value;

        value = 16;

        System.out.println("16 decimal = " + value);

        value = 020;

        System.out.println("20 octal = " + value + " en decimal".);

        value = 0x10;

      System.out.println("10 hexadecimal = " + value + " en decimal".);
    }
}
```

Esto es lo que el programa visualiza:

```
C:\>java app
16 decimal = 16
20 octal = 16 en decimal.
10 hexadecimal = 16 en decimal.
```

Creación de literales en coma flotante

El programador novato aparece y dice, "Tengo un problema. Quiero poner un número en coma flotante, 1.5, en una variable de coma flotante, pero Java insiste en decir "Tipo incompatible para =. Se necesita conversión específica para *double*". ¿Qué ocurre?" "Por defecto", le dice, "los números en coma flotante que se utilizan como literales son de tipo *double*, no de tipo *float*. Se pueden cambiar añadiendo una 'f' o 'F' al final del literal para convertirlo en *float* o una 'd' o 'D' para pasarlo a *double*". "¡Ah!", contesta el PN.

En el código, los literales en coma flotante son de tipo *double* por defecto, como por ejemplo, 3.1415926535, 1.5 y 0.1111111. La notación estándar para los literales en coma flotante es un número seguido por una parte fraccionaria. Además se puede indicar en potencia de 10 con 'e' o 'E':

```
1.345E10
```

Esto es lo mismo que 1.345 x 10^{10} ó –9.999E-23, que es lo mismo que –9.999 x 10^{-23}.

A continuación, vemos un ejemplo en el que se trata de asignar un literal en coma flotante a una variable tipo *float*:

```
public class app
{
    public static void main(String[] args)
    {
        float value;

        value = 1.5f;

        System.out.println("El valor = " + value);
    }
}
```

Desafortunadamente, el tipo por defecto de los literales en coma flotante, es *double*, por lo que Java devuelve el siguiente error:

```
C:\>javac app.java -deprecation
app.java:7: Incompatible type for =. Explicit cast needed to convert
double
to float.
        value = 1.5;
                ^
1 error
```

Esto se puede arreglar explícitamente convirtiendo el literal a *float*, como se indica a continuación:

```
public class app
{
    public static void main(String[] args)
    {
        float value;
    value = 1.5f;
        System.out.println("El valor = " + value);
    }
}
```

Ahora, el código se ejecuta como esperamos:

```
C:\>java app
El valor = 1.5
```

Creación de literales booleanos

En Java, los valores booleanos sólo pueden ser verdaderos o falsos (no 0 ó 1 u otros valores numéricos como ocurre en otros lenguajes; esto forma parte de la insistencia de Java en los tipos de datos). Esto quiere decir que los únicos dos literales booleanos que se pueden usar son verdadero y falso.

A continuación vemos un ejemplo que utiliza verdadero *(true)* como literal booleano:

```
public class app
{
    public static void main(String[] args)
    {
        boolean value;
  value = true;
        System.out.println("El valor = " + value);
    }
}
```

Este es el resultado del programa:

```
C:\>java app
El valor = true
```

Creación de literales carácter

"Hey", dice el programador novato, "¿cómo asigno una letra a una variable en Java? Estoy evaluando todos los productos de la empresa y quiero asignarles letras". "Puede usar literales de tipo carácter, cada uno de los cuales representa un carácter", le responde. "Por cierto, ¿el gran jefe sabe esto?" "Todavía no", dice el PN.

La forma básica de un literal de tipo carácter en Java es un valor que corresponde a un carácter del conjunto de caracteres Unicode (para más detalles sobre Unicode, ver www.unicode.org). Los literales de tipo carácter son números que actúan como índices dentro del conjunto de caracteres Unicode, no caracteres actuales. Por ejemplo, el código Unicode para la letra 'C' es 67. Sin embargo, la siguiente aplicación visualiza una 'C':

```
public class app
{
    public static void main(String[] args)
    {
        char char3;

  char3 = 67;
```

```
        System.out.println("La tercera letra del alfabeto = " + char3);
    }
}
```

También, puede referirse al código Unicode para la letra C con un literal de tipo carácter encerrado entre comillas simples:

```
public class app
{
    public static void main(String[] args)
    {
        char char3;

        char3 = 'C';

        System.out.println("La tercera letra del alfabeto = " + char3);
    }
}
```

Además de encerrar los caracteres entre comillas simples, también puede encerrar secuencias de escape entre este tipo de comillas para obtener literales de tipo carácter que no pueden conseguirse tecleando un simple carácter. Estas son las secuencias de escape:

- \' (comilla simple)
- \" (comillas dobles)
- \\ (barra invertida)
- \b (espacio en blanco)
- \ddd (carácter octal)
- \f (avance)
- \n (nueva línea)
- \r (retorno de carro)
- \t (tabulador)
- \uxxxx (carácter Unicode en hexadecimal)

Por ejemplo, si quiere que se visualice una doble comilla que aparece en el texto, puede utilizar la secuencia de escape \":

```
public class app
{
    public static void main(String[] args)
    {
  System.out.println("Él dijo, \"¡Hola!\"");

    }
}
```

Este es el resultado del código:

```
C:\>java app
Él dijo, "¡Hola!"
```

Creación de literales tipo cadena

El programador novato regresa con un café. "De acuerdo", dice, "este es el problema: quiero usar una sentencia println para sacar varias líneas, ¿puedo hacerlo?" "Claro", le dice, "utilice el literal de tipo carácter \n para añadir una nueva línea". "¿Cómo?, pregunta PN.

Esto es un ejemplo de lo que PN quiere hacer. En este caso, visualizaré varias líneas de texto utilizando el literal \n para comenzar una nueva línea:

```
public class app
{
    public static void main(String[] args)
    {
  System.out.println("Esto es \nun texto\nde varias líneas".);
    }
}
```

Esta es la salida de la aplicación:

```
C:\>java app
Esto es
un texto
de varias líneas
```

Como con la mayoría de los lenguajes de programación, se pueden poner cadenas entre comillas dobles (al igual que los literales de tipo carácter se encierran entre comillas simples). Además se pueden utilizar secuencias de escape introducidas en la sección anterior. Observe que el compilador convierte los literales de tipo cadena en objetos *String*, no en tipos de datos sencillos inherentes (lo que quiere decir que el código como "Hola".length() es perfectamente legal y devuelve la longitud de la cadena "Hola").

Declaración de variables de tipo entero

"Realmente, ahora es cuando estoy programando", dice el programador novato, "y necesito almacenar algunos datos enteros. ¿Cómo puedo hacerlo?" "Con una variable entera", le dice. "Tome una silla y veámoslo".

Java utiliza cuatro tipos de enteros, cada uno con su propio número de bytes alocados en memoria: *byte* (un byte), *short* (dos bytes), *int* (cuatro

bytes) y *long* (ocho bytes). Para conocer el posible rango de valores que cada uno puede tomar, ver el punto "¿De qué tipos de datos disponemos?", en este capítulo. Usar uno u otro depende del rango de datos que se quiera, así como de otras consideraciones, como cuánta memoria hay disponible (en caso de que se quieran tener muchos enteros).

Esto es un ejemplo que utiliza todos los tipos enteros, declara un entero de cada tipo, asigna a cada tipo un dato y luego visualiza ese dato:

```
public class app
{
    public static void main(String[] args)
    {
        byte byte1;
        short short1;
        int int1;
        long long1;
        byte1 = 1;
        short1 = 100;
        int1 = 10000;
        long1 = 100000000;

        System.out.println("byte1 = " + byte1);
        System.out.println("short1 = " + short1);
        System.out.println("int1 = " + int1);
        System.out.println("long1 = " + long1);
    }
}
```

Este es el resultado de la aplicación:

```
byte1 = 1
short1 = 100
int1 = 10000
long1 = 100000000
```

Declaración de variables de tipo coma flotante

"Lo siento", dice el programador novato, "pero los enteros no se truncan. Estoy tratando de diseñar un conversor de moneda y creía que se podía ignorar la parte decimal de cada valor, pero el gran jefe me dijo que todos los decimales cuentan. ¿Hay algún otro tipo de datos que pueda utilizar?" "Claro", le dice. "Puede usar *floats* y *doubles*".

Java incluye dos tipos de variables en coma flotante, cada uno con su propio número de *bytes* para alocar en memoria: *float* (cuatro bytes) y *double* (ocho bytes). Para ver el rango posible de valores que cada tipo puede gestionar, ver la sección "¿De qué tipo de datos disponemos?", en este capítulo. El

usar uno u otro depende del rango de los datos que se quiera, así como de otras consideraciones, como cuánta memoria hay disponible (en el caso de que se quiera tener muchos valores en coma flotante).

A continuación tiene un ejemplo que declara y usa ambos tipos, *float* y *double* (observe que explícitamente hago que los literales sean *float* o *double*, por lo que no habrá ningún problema con las conversiones de tipo):

```
public class app
{
    public static void main(String[] args)
    {
        float float1;
        double double1;

        float1 = 1.11111111111F;
        double1 = 1.1111111111111E+9D;

        System.out.println("float1 = " + float1);
        System.out.println("double1 = " + double1);
    }
}
```

Esta es la salida del código (observe que he sobrepasado la precisión permitida para un float, por lo que el valor se ha redondeado):

```
C:\>java app
float1 = 1.1111112
double1 = 1.1111111111111E9
```

Declaración de variables de tipo carácter

Se pueden declarar variables de tipo carácter con la palabra clave *char*. Para los posibles valores que se pueden almacenar en una variable *char*, ver también en este capítulo.

Esto es un ejemplo en el que se declaran dos variables de tipo *char*: char1 y char2. Este ejemplo prueba que se puede asignar a un *char* un código Unicode o un literal de tipo carácter (de hecho, el compilador traduce los literales de tipo carácter en códigos Unicode):

```
public class app
{
    public static void main(String[] args)
    {
        char char1, char2;
    char1 = 65;
    char2 = 'B';

        System.out.println("char1 = " + char1);
```

```
        System.out.println("char2 = " + char2);
    }
}
```

Este es el resultado del código:

```
C:\>java app
char1 = A
char2 = B
```

A continuación veremos un punto clave que se tratará en un tema futuro: añadir parte de un texto al final del char1, convirtiéndolo a una cadena, e incrementar el valor de char2, cambiando de 'B' a 'C':

```
public class app
{
    public static void main(String[] args)
    {
        char char1, char2;

        char1 = 65;
        char2 = 'B';

        System.out.println("char1 = " + char1);
        System.out.println("char2 = " + char2);
        System.out.println("char1 + 1 = " + char1 + 1);
        System.out.println("++char2 = " + ++char2);
    }
}
```

Esta es la salida de la nueva versión del programa:

```
char1 = A
char2 = B
char1 + 1 = A1
++char2 = C
```

Declaración de variables de tipo booleano

Las variables de tipo booleano se declaran con la palabra clave *boolean*. En Java, las variables de tipo booleano sólo pueden tener dos valores: verdadero y falso (no valores numéricos como 0 y 1, como ocurre en otros lenguajes de programación).

Aquí tenemos un ejemplo en el que se declara y se usan dos variables de tipo booleano:

```
public class app
{
    public static void main(String[] args)
```

```
    {
  boolean boolean1, boolean2;

  boolean1 = true;
  boolean2 = false;

  System.out.println("boolean1 = " + boolean1);

  System.out.println("boolean2 = " + boolean2);

    }
}
```

Este es el resultado del código:

```
C:\>java app
boolean1 = true
boolean2 = false
```

Generalmente, durante las pruebas, se usan valores booleanos para determinar el flujo de un programa. De momento, no diré más sobre ello y le daremos un repaso en el siguiente capítulo. En este caso, estoy usando dos tipos de variables con la sentencia *if* de Java. Compruebo el valor de boolean1 con la sentencia *if*, haciendo que el código visualice el mensaje "boolean1 es verdadero"., si es verdadero (*true*) y "boolean1 es falso". en caso contrario:

```
public class app
{
    public static void main(String[] args)
    {
        boolean boolean1, boolean2;

        boolean1 = true;
        boolean2 = false;

        System.out.println("boolean1 = " + boolean1);

        System.out.println("boolean2 = " + boolean2);

  If(boolean1) {
      System.out.println("boolean1 es verdadero".);
  } else {
      System.out.println("boolean1 es falso".);
  }
    }
}
```

El nuevo resultado es:

```
C:\>java app
boolean1 = true
boolean2 = false
boolean1 es verdadero.
```

Inicialización de variables

"De acuerdo", dice el programador novato, "Ya lo tengo todo claro. Primero, declaro una variable y después le asigno un valor". "Realmente", le dice, "puede hacer las dos cosas en un solo paso". El PN contesta, "¡dígame cómo!"

Pues bien, vamos a declarar variables y a asignarles valores de la siguiente forma:

```
public class app
{
    public static void main(String[] args)
    {
  int int1;

  int1 = 1;

        System.out.println("int1 = " + int1);
    }
}
```

Sin embargo, puedo combinar estos dos pasos en uno solo inicializando una variable cuando la declaro:

```
public class app
{
    public static void main(String[] args)
    {
  int int1 = 1;

        System.out.println("int1 = " + int1);
    }
}
```

A continuación se indica cómo se declaran e inicializan varias variables:

```
public class app
{
    public static void main(String[] args)
    {
  int int1 = 1, int2 = 2, int3 = 3;

         System.out.println("int1 = " + int1 + ", int2 = " + int2 +
        ", int3 = " + int3);
    }
}
```

Este es el resultado del programa:

```
C:\>java app
int1 = 1, int2 = 2, int3 = 3
```

Inicialización dinámica

Llegados a este punto, acabamos de asignar valores constantes a variables, pero se puede asignar cualquier expresión (una expresión es cualquier combinación de términos Java que da un valor) a una variable cuando es declarada, siempre que la expresión sea válida en ese momento. Por ejemplo, aquí se está asignando el valor 2 a int1 y 3 a int2 y el valor de int1 veces int2 a int3, usando el operador multiplicación de Java (*):

```
public class app
{
    public static void main(String[] args)
    {
        int int1 = 2, int2 = 3;
        int int3 = int1 * int2;

        System.out.println("int1 = " + int1 + ", int2 = " + int2 +
        ", int3 = " + int3);
    }
}
```

Esto es lo que nos devuelve el código cuando lo ejecutamos:

```
C:\>java app
int1 = 2, int2 = 3, int3 = 6
```

Observe que el compilador de Java no tiene ni idea de lo que valdrá int1 veces int2 cuando crea los *bytecodes* para esta aplicación. Esto significa que el valor actual con el que int3 está inicializado será determinado en tiempo de ejecución, y es por lo que este proceso se llama inicialización dinámica.

Al igual que en C++, en Java se puede introducir declaración de variables en el código, como se indica a continuación:

```
public class app
{
    public static void main(String[] args)
    {
  int int1 = 2, int2 = 3;

     System.out.println("int1 = " + int1 + ", int2 = " + int2);

  int int3 = int1 * int2;

     System.out.println("int3 = " + int3);
    }
}
```

Este es el resultado del código:

```
C:\>java app
int1 = 2, int2 = 3
int3 = 6
```

Conversión de tipos de datos

"Uf", dice el programador novato. "Me he atascado. Tengo una variable de tipo *int* y quiero asignarla a una variable de tipo *byte*, pero Java me devuelve un error "Tipo incompatible para =". ¿Qué es lo que está mal?" "Es un problema de conversión de tipos", le explica, "y tiene que usar explícitamente un *cast* de tipos". "Hmm", dice PN, "¿cómo se hace eso?"

Java es un lenguaje muy insistente con los tipos, y como consecuencia de ello, con frecuencia nos enfrentamos a la situación de asignar una variable de un tipo a una variable de otro. Hay dos formas de hacerlo: contando con una conversión automática y haciendo explícitamente un *cast* de tipos. Veámoslas.

Conversiones automáticas

Cuando asigna un tipo de dato a una variable de otro tipo, Java convertirá el dato al nuevo tipo de variable de forma automática si las dos condiciones siguientes son verdaderas:

- Los tipos del dato y de la variable son compatibles.
- El tipo de destino tiene un rango mayor que el de origen.

Por ejemplo, se puede asignar un valor de tipo *byte* a una variable *int*, ya que *byte* e *int* son tipos compatibles y las variables *int* tienen un rango mayor que los valores *byte*. Por lo tanto, no se perderá ningún dato en la conversión de tipos. Esto es un ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
        byte byte1 = 1;
        int int1;

        int1 = byte1;

        System.out.println("int1 = " + int1);
```

```
    }
}
```

El compilador de Java no tiene ningún problema con este código y hace la conversión de tipos automáticamente. Este es el resultado del programa:

```
C:\>java app
int1 = 1
```

Este tipo de conversiones, en las que se convierte a un tipo de datos que tiene mayor rango, se llaman extensión de conversiones. En ellas, los tipos numéricos, como entero o coma flotante, son compatibles entre sí. Por otro lado, los tipos *char* y *boolean* no son compatibles entre sí y tampoco con los tipos numéricos.

Casting a nuevos tipos de datos

Si se está asignando un valor que es de un tipo con un rango mayor que la variable a la que se le está asignando, se está ejecutando lo que se denomina estrechamiento de conversiones. El compilador de Java no las ejecuta automáticamente, ya que se perdería la posibilidad de precisión.

Si se quiere hacer un estrechamiento de conversiones, se debe usar explícitamente un *cast*, que tiene el siguiente formato:

```
(tipo de dato de destino) valor
```

Por ejemplo, en este código, se está convirtiendo un tipo entero a un tipo byte:

```
public class app
{
    public static void main(String[] args)
    {
        byte byte1;
        int int1 = 1;

  byte1 = (byte) int1;

        System.out.println("byte1 = " + byte1);
    }
}
```

Si no se hace explícitamente el *cast*, el compilador devolverá error, pero con el *cast* de tipos, no hay problema, ya que Java decide que se conoce la posibilidad de perder algunos datos cuando se introduce un valor probablemente mayor en un tipo más pequeño. Por ejemplo, cuando se pone un número en coma flotante en un *long*, la parte fraccional del número se trunca-

rá, y puede que se pierdan más datos si el valor en coma flotante está fuera del rango que un *long* puede gestionar. Esta es la salida del código:

```
C:\>java app
byte1 = 1
```

Algo a tener en cuenta es que el compilador de Java convierte a un tipo de mayor precisión, si es necesario, al evaluar expresiones. Por ejemplo, vamos a considerar el siguiente código, en el que parece que sólo hay bytes involucrados:

```
public class app
{
    public static void main(String[] args)
    {
        byte byte1 = 100;
        byte byte2 = 100;
        byte byte3;

        byte3 = byte1 * byte2 /100;

        System.out.println("byte3 = " + byte3);
    }
}
```

Por lo tanto, como Java sabe que de la multiplicación de tipos puede resultar valores del tamaño de un entero, automáticamente convierte el resultado de byte1 * byte2 en un entero, lo que quiere decir que realmente hay que usar explícitamente un *cast* para mantener el tipo byte:

```
public class app
{
    public static void main(String[] args)
    {
        byte byte1 = 100;
        byte byte2 = 100;
        byte byte3;

        byte3 = (byte) (byte1 * byte2 / 100);

        System.out.println("byte3 = " + byte3);
    }
}
```

Este código se compila y se ejecuta como esperábamos, pero no sería así sin el *cast* (byte):

```
C:\>java app
byte3 = 100
```



Truco: en general, el compilador de Java convierte, en las expresiones, *bytes* y *shorts* a *int*. Si un operando es *long*, la expresión entera se convierte a *long*. De igual manera, si un operando es *float*, toda la expresión será *float*; si un operando es *double*, toda la expresión es *double*.

Declaración de *arrays* unidimensionales

El gran jefe aparece y dice, "Es hora de atacar a los clientes que nos deben facturas". "De acuerdo", le dice, "¿puedo ver las cuentas?" "Realmente, nunca guardamos las cuentas", dice el GJ. "Ah", le contesta. "Creo que tendré que crear un *array* para almacenar las cuentas".

Como ya vimos antes en este capítulo, los *arrays* proporcionan una forma fácil de gestionar un conjunto de datos por medio de un índice, lo que es fácil para los ordenadores, ya que se puede manipular el índice en el código. Java soporta *arrays* unidimensionales y multidimensionales, y ambos los veremos aquí.

Para tener un *array* preparado es necesario ejecutar dos pasos. Primero, se debe declarar el *array*. A continuación se indica cómo se declara, en general, un array unidimensional:

```
tipo nombre[];
```

Por ejemplo, así es como se declara un *array* de valores *double*, que se llama accounts[]:

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[];
        .
        .
        .
    }
}
```



Truco: de hecho, hay otra forma de hacer esto siguiendo la sintaxis de la declaración de punteros en C++. Además se pueden declarar *arrays* con corchetes ([]) después del tipo, no del nombre de la variable; por ejemplo: double[] accounts.

Al igual que al declarar variables sencillas, la declaración de un *array* no reserva memoria, ya que Java no sabe exactamente qué tamaño va a tener. Esto quiere decir que es necesario otro paso en este proceso: la creación del *array*. Veamos el siguiente punto para más detalles.

Creación de arrays unidimensionales

Después de que se ha declarado un array unidimiensional, el siguiente paso es crear ese array alocando memoria para él. Como se verá en el capítulo o usar el nuevo array de la siguiente forma:

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[];

        accounts = new double[100];

        accounts[3] = 1335.67;

        System.out.println("La cuenta 3 debe $" + accounts[3]);
    }
}
```

Aquí se ha creado un array de exactamente 100 valores double, que Java inicializa a 0. El límite inferior de todo *array* de Java es 0, por lo que el primer elemento del *array* es accounts[0] y el superior es accounts[99]. Si el índice del *array* está fuera del rango del 0 al 99, Java devolverá un error fatal, y el programa se parará.

Este es el resultado del programa:

```
C:\>java app
La cuenta 3 debe $1335.67
```

De hecho, se puede combinar el proceso de declaración y creación de *arrays* en un paso:

```
public class app
{
    public static void main(String[] args)
    {

        double accounts[ ] = new double [100];

        accounts[3] = 1335.67;

        System.out.println("La cuenta 3 debe $" + accounts[3]);
```

```
    }
}
```

Inicialización de *arrays* unidimensionales

El programador novato regresa con una pregunta. "Sé que puedo inicializar variables sencillas cuando las declaro", dice, "pero ¿puedo hacer lo mismo con los *arrays*?" "Sin problema", le dice.

Para inicializar los arrays unidimensionales, únicamente hay que poner los valores entre llaves, un valor detrás de otro, separados por comas, empezando con el primer valor del *array*. Este es un ejemplo que inicializa los primeros cuatro elementos del *array* accounts[ ] con datos:

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[] = {238.45, 999.33, 0, 1335.67};

        accounts[3] = 1335.67;

        System.out.println("La cuenta 3 debe $" + accounts[3]);
    }
}
```

Declaración de *arrays* multidimensionales

"Hmm", dice pensativo el programador novato, "creo que necesito más que un *array* unidimensional. Estoy pensando en mantener los productos indexados por número de producto, y el *array* supongo que almacenará el número de artículos del inventario, su coste, las ventas, el número"... "Manténgalo", le dice. "Utilice un *array* multidimensional".

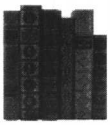
Se pueden declarar *arrays* multidimensionales de la misma forma que se declaran los unidimensionales; sólo con incluir un par de corchetes para cada dimensión del *array*.

```
tipo nombre[][][]...;
```

Ya vimos anteriormente, en este capítulo, cómo declarar un *array* de dos dimensiones con dos filas y 100 columnas:

```
public class app
{
    public static void main(String[] args)
```

```
    {
        double accounts[][] = new double[2][100];
        .
        .
        .
    }
}
```



Información: de hecho, hay otra forma de hacer esto, siguiendo la sintaxis de C++ para la declaración de punteros. Además, también se pueden declarar *arrays* con los corchetes ([]) después del tipo, no del nombre de la variable: double[][] accounts.

Así funcionan los *arrays* de dos dimensiones: el índice izquierdo especifica la fila y el derecho, la columna.

Por supuesto, no tiene por qué limitarse a las dos dimensiones, aquí tiene cómo declarar un *array* de cuatro dimensiones:

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[][][][] = new double[2][3][4][5];
        .
        .
        .
    }
}
```

Como puede observar, es tan fácil declarar *arrays* multidimensionales como declararlos unidimensionales. Ahora, ¿cómo se crea un *array* después de haberlo declarado? Ver el siguiente punto para más detalles.

Creación de *arrays* multidimensionales

El programador novato pregunta, "Ahora que he declarado un nuevo *array* multidimensional, ¿cómo lo creo?" "Veámoslo", le dice.

Con el operador *new* se crea un nuevo *array* multidimensional reservándole memoria y dándole las dimensiones que se quiera. Veámoslo con un ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
```

```
        double accounts[][];

        accounts = new double[2][100];

        accounts[0][3] = 43.95;
        accounts[1][3] = 2385489382.06;

      System.out.println("La cuenta de ahorro 3 tiene $" + accounts[0][3]);
            System.out.println("La  cuenta  corriente  3  tiene  $"  +
accounts[1][3]);
      }
  }
```

Este es el resultado del código:

```
C:\>java app
La cuenta de ahorro 3 tiene $43.95
La cuenta corriente 3 tiene $2.38548938206E9
```

Además, la declaración y reserva de memoria se pueden hacer en un solo paso:

```
  public class app
  {
      public static void main(String[] args)
      {
          double accounts = new double[2][100];

          accounts[0][3] = 43.95;
          accounts[1][3] = 2385489382.06;

      System.out.println("La cuenta de ahorro 3 tiene $" + accounts[0][3]);
            System.out.println("La  cuenta  corriente  3  tiene  $"  +
accounts[1][3]);
      }
  }
```

En este ejemplo se crea y utiliza un *array* de cuatro dimensiones:

```
  public class app
  {
      public static void main(String[] args)
      {
    double accounts[][][][] = new double[2][3][4][5];
       accounts[0][1][2][3] = 43.95;

    System.out.println("La cuenta [0][1][2][3] tiene $" +
    accounts[0][1][2][3]);
      }
  }
```

Este es el resultado del programa:

```
C:\>java app
La cuenta [0][1][2][3] tiene $43.95
```

Los *arrays* multidimensionales son realmente *arrays* de *arrays*, lo que significa que si se tiene un *array* de dos dimensiones (array[][]), se puede tratar como un *array* de *arrays* unidimensionales, al que se puede acceder con array[0], array[1], array[2] y así sucesivamente. Hay una manera sencilla de hacer esto utilizando un bucle *for* (del que veremos más en el siguiente capítulo) y el método *length* (que veremos en las siguientes secciones) para averiguar la longitud de un *array*:

```
public class app
{
    public static void main(String[] args)
    {
        double array[][] = {{1, 2, 3},
                            {3, 2, 1},
                            {1, 2, 3}};
        int sum = 0, total = 0;

        for(int outer_index = 0; outer_index < array.length;
            outer_index++) {
            for(int inner_index = 0; inner_index <
                array[outer_index].length; inner_index++) {

                sum += array[outer_index][inner_index];
                total++;

            }
        }

        System.out.println("Valor medio del array = " + (sum / total));
    }
}
```

Este es el resultado del código:

```
C:>\java app
Valor medio del array = 2
```

Hasta ahora, todos los *arrays* que hemos utilizado tenían el mismo número de elementos en cada dimensión, pero no es necesario que sea así. Para aprender más, se puede echar un vistazo a la sección que viene después de la siguiente. Primero, veamos cómo se inicializan los *arrays* multidimensionales.

Inicialización de *arrays* multidimensionales

Los *arrays* multidimensionales se pueden inicializar cuando se los declara, de la misma forma que se inicializan los unidimensionales; basta con incluir un par de corchetes para cada dimensión y poner los valores con los

que se quiere inicializar el *array* dentro de los mismos. Por ejemplo, aquí vemos cómo se inicializa un *array* de dos dimensiones:

```
public class app
{
    public static void main(String[] args)
    {
        double accounts[ ][ ] = {{10.11, 19.56, 4343.91, 43.95},
                                 {11.23, 54.23, 543.62,
2385489382.06}};

        System.out.println("La cuenta de ahorro 3 tiene $" +
            accounts[0][3]);

        System.out.println("La cuenta corriente 3 tiene $" +
            accounts[1][3]);
    }
}
```

La ejecución del código produce:

```
C:\>java app
La cuenta de ahorro 3 tiene $43.95
La cuenta corriente 3 tiene $2.38548938206E9
```

Creación de *arrays* multidimensionales

"De acuerdo", dice orgulloso el progamador novato, "ya soy un experto en *arrays*". "Ah, ¿sí?", le dice. "¿Sabe cómo poner diferente número de elementos en cada fila de un *array*?" El PN dice, "¿disculpe?" Como en otros muchos lenguajes de programación, en Java, los *arrays* multidimensionales son *arrays* de *arrays*. Esto quiere decir que se pueden construir *arrays* como se quiera; como en el ejemplo, en el que cada fila de un *array* de dos dimensiones tiene un diferente número de elementos:

```
public class app
{
    public static void main(String[] args)
    {
        double array[ ][ ] = new double[5][ ];

        array[0] = new double[500];
        array[1] = new double[400];
        array[2] = new double[300];
        array[3] = new double[200];
        array[4] = new double[100];

        array[3][3] = 1335.67;

        System.out.println("La cuenta [3][3] tiene $" + array[3][3]);
```

```
    }
}
```

Lo que ocurre aquí es que se está tratando cada fila de un *array* de dos dimensiones como un *array* unidimensional y creando cada uno de esos *arrays* unidimensionales de forma separada. Este es el resultado del código:

```
C:\>java app
La cuenta [3][3] tiene $1335.67
```

Longitud de un *array*

Con frecuencia, es útil conocer la longitud de un *array*, especialmente si se está iterando sobre todos los elementos del *array* dentro del código. Para saber el número de elementos de un *array* llamado array1, se puede usar el término *array1.length*. Este es un ejemplo del capítulo siguiente, que utiliza el bucle *for* para calcular el grado medio de un estudiante entre un grupo de seis grados (en este caso, *grades.length* devuelve el valor 6):

```
public class app
{
    public static void main(String[] args)
    {
        double grades[] = {88, 99, 73, 56, 87, 64};
        double sum, average;

        sum = 0;

        for (int loop_index = 0; loop_index < grades.length;
            loop_index++) {
            sum += grades[loop_index];
        }

        average = sum / grades.length;

        System.out.println("Grado medio = " + average);
    }
}
```

Este es el resultado:

```
C:\>java app
Grado medio = 77.83333333333333
```

La clase *String*

"He estado revisando la lista de tipos de datos sencillos de Java", dice el programador novato, "y no encuentro las cadenas de texto. ¿No deberían

estar ahí?" "Algunas personas dicen que sí, " le responde, "pero, de hecho, en Java, las cadenas se gestionan como si fueran objetos. Una de las ventajas de esto es que un objeto de tipo *string* tiene gran variedad de métodos que se pueden usar".

En muchos lenguajes, las cadenas de texto son tipos de datos fundamentales inherentes al lenguaje, pero en Java, las cadenas son gestionadas con las clases *String* y *StringBuffer*. Veamos primero la clase *String*.

Los objetos de tipo *string* gestionan las cadenas de texto que no se pueden cambiar; si se quiere cambiar el texto actual de la cadena, se debería usar la clase *StringBuffer*. Este es un ejemplo en el que se crea una cadena y se visualiza (observe cómo este código hace la clase *String* como si fuera otro tipo de dato sencillo):

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        System.out.println(s1);
    }
}
```

Este es el resultado:

```
C:\>java app
¡Hola desde Java!
```

La clase *String* es muy poderosa, pues con los métodos que proporciona permite convertir la cadena en un *array* de caracteres, convertir números en cadenas, buscar cadenas, crear *substrings*, cambiar la cadena de mayúsculas a minúsculas o viceversa, obtener la longitud de la cadena, comparar cadenas y mucho más.

La clase *String* es una clase, no un tipo de dato intrínseco, lo que significa que se crean objetos de esa clase con constructores, de los que aprenderá todo lo necesario en el capítulo 4. Un constructor es como un método normal de una clase, salvo cuando se usa para crear un objeto de esa clase. Aquí verá una rápida presentación de los constructores de la clase *String*. La clase *String* tiene, además, un miembro de datos que se usa cuando se comparan cadenas (lo veremos en el siguiente capítulo). Este miembro de datos se muestra en la tabla 2.2. Los constructores de la clase *String* que se pueden usar para crear objetos *String* (ver la sección "Creación de cadenas", en este capítulo), aparecen en la tabla 2.3 y los métodos de la clase *String* aparecen en la tabla 2.4.

Estas tablas se usarán en las siguientes secciones, en las que crearemos y usaremos objetos de *String*.

Tabla 2.2. Resumen de los campos de la clase *String*.

Campo	Descripción
Static Comparator CASE_INSENSITIVE_ORDER	Permite la ordenación de objetos *String*, en comparación con TolgnoreCase.

Tabla 2.3. Resumen de los constructores de la clase *String*.

Constructor	Descripción
String( )	Inicializa un nuevo objeto *String* para gestionar un secuencia de caracteres vacía.
String(byte[ ] bytes)	Construye un nuevo objeto *String* convirtiendo el array de *bytes* mediante la codificación de caracteres que tiene la plataforma por defecto.
String(byte[ ] ascii, int hibyte)	Obsoleto. Este método no convierte bien los *bytes* en caracteres.
String(byte[ ] bytes, int offset, int length)	Construye un nuevo objeto String convirtiendo el subarray de bytes usando la codificación de caracteres por defecto.
String(byte[ ] ascii, int hibyte, int offset, int count)	Obsoleto. Este método no convierte bien bytes en caracteres.
String (byte[ ] bytes, int offset, enc)	Construye un nuevo objeto *String* convirtiendo el entero *length* y el subarray de caracteres utilizando la codificación de caracteres especificada.
String (byte[ ] bytes, String enc)	Construye un nuevo objeto *String* convirtiendo el *array* de *bytes* mediante la codificación de caracteres que se especifica.
String(char[ ] value)	Aloca un nuevo objeto *String* para representar la secuencia de carac-

Constructor	Descripción
	teres contenida en el argumento *array* de caracteres.
String(char[ ] value, int offset, int count)	Aloca un nuevo objeto *String* que contiene caracteres de un subarray del array de caracteres del argumento.
String(String value)	Inicializa un nuevo objeto *String* para que represente la misma secuencia de caracteres que el argumento.
String(StringBuffer buffer)	Aloca un nuevo objeto *String* que contiene la secuencia de caracteres del argumento *buffer*.

Tabla 2.4. Métodos de la clase *String*.

Método	Descripción
char charAt(int index)	Proporciona el carácter del índice especificado.
int compareTo(Object o)	Compara este objeto *String* con otro.
int compareTo(String anotherString)	Compara lexicográficamente dos cadenas.
int compareTolgnoreCase(String str)	Compara lexicográficamente dos cadenas ignorando mayúsculas o minúsculas.
String concat(String str)	Concatena la cadena dada al final de esta cadena.
Static String copyValueOf(char[ ] data)	Produce un objeto de tipo String que es equivalente al array de caracteres dado.
Static String copyValueOf(char[ ] it offset, int count)	Produce un objeto de tipo String con datos equivalentes al array de caracteres dado, usando offsets.
boolean endsWith(String sufijo)	Verdadero si esta cadena termina con el sufijo dado.

Método	Descripción
boolean equals(Object un objeto)	Compara esta cadena con un objeto.
boolean equalsIgnoreCase(String otra cadena)	Compara este objeto String con otro objeto, ignorando las mayúsculas.
byte[ ] getBytes	Convierte este objeto *String* en *bytes,* de acuerdo con la codificación de caracteres por defecto y almacenando el resultado en un nuevo *array* de *bytes.*
void getBytes(int srcBegin, int srcEnd, byte[ ] dst, int dstBegin)	Obsoleto. Este método no convierte bien caracteres a bytes.
byte[ ] getBytes(String enc)	Convierte este objeto *String* en *bytes* de acuerdo con la codificación de caracteres dada, almacenando el resultado en un nuevo *array* de *bytes.*
void getBytes(int srcBegin, int srcEnd, char[ ] dst, int dstBegin)	Copia caracteres de esta cadena en el array de destino.
int hashCode()	Produce un *hashcode* para esta cadena.
int indexOf(int ch)	Proporciona el índice, dentro de la cadena, de la primera vez que aparece el carácter dado.
int indexOf(int ch, int fromIndex)	Produce el índice, dentro de la cadena, de la primera vez que aparece el carácter dado empezando por el índice especificado.
int indexOf(Sting str)	Proporciona el índice, dentro de la cadena de la primera vez que aparece el *substring* especificado.
int indexOf(String str, int fromIndex)	Produce el índice, dentro de la cadena, de la primera vez que aparece el *substring* dado, empezando por el índice especificado.
String intern( )	Produce una representación para el objeto *String.*

Método	Descripción
int lastIndexOf(int ch)	Devuelve el índice, dentro de la cadena, de la última ocurrencia del carácter especificado.
int lastIndexOf(int ch, int fromIndex)	Produce el índice, dentro de la cadena de la última ocurrencia del carácter dado, buscando hacia atrás desde el índice especificado.
int lastIndexOf(String str)	Devuelve el índice, dentro de la cadena, de la ocurrencia más a la derecha del *substring* dado.
int lastIndexOf(String str, int fromIndex)	Produce el índice dentro de esta cadena de la última ocurrencia del *substring* dado.
int length()	Devuelve la longitud de la cadena
boolean regionMatches String other, (boolean ignoreCase, int toffset, int ooffset, int len)	Comprueba si dos regiones de cadenas son iguales, permitiendo ignorar las mayúsculas.
boolean regionMatches(int toffset, String other, int ooffset, int len)	Chequea si dos regiones de cadenas son iguales.
String replace (char oldChar, char newChar)	Produce una nueva cadena situando todas las ocurrencias de oldChar en esta cadena con newChar.
boolean startsWith(String prefix)	Chequea si esta cadena empieza con el prefijo dado.
boolean startsWith (String prefix, int toffset)	Chequea si esta cadena empieza con el prefijo dado empezando con el índice dado.
String substring(int beginIndex)	Produce una nueva cadena que es una subcadena de esta.
char[ ] to CharArray( )	Convierte esta cadena en un nuevo *array* de caracteres.
String toLowerCase( )	Convierte a minúsculas todos los caracteres de este objeto *String,* usando las reglas del defecto local, que es devuelto por Locale.getDefault.

Método	Descripción
String toLowerCase(Locale locale)	Convierte a minúsculas todos los caracteres de este objeto *String*, usando las regalas del argumento locale especificado.
String toString( )	Se devuelve este objeto (que ya es una cadena).
String toUpperCase( )	Convierte a mayúsculas todos los caracteres de este objeto *String*, utilizando las reglas del locale por defecto, que es devuelto por Locale.getDefault.
String toUpperCase (Locale locale)	Convierte a maýusculas todos los caracteres de este objeto *String*, usando las reglas del locale dado.
String trim( )	Elimina el espacio en blanco desde ambos finales de esta cadena.
Static String valueOf(boolean b)	Produce la representación de la cadena del argumento booleano.
Static String valueOf(char c)	Produce la representación de la cadena del argumento *char*.
Static String valueOf(char[ ] data)	Produce la representación de la cadena del *array* de caracteres que se pasa como argumento.
Static String valueOf(char[ ] data, offset, int count)	Produce la representación de la cadena de un subarray específico del argumento*char array*de enteros.
Static String valueOf(double d)	Produce la representación *string* de un *double*.
Static String valueOf(float f)	Produce la representación *string* de un *float*.
Static String valueOf(int i)	Produce la representación *string* de un *int*.
Static String valueOf(long l)	Produce la representación *string* de un *long*.
Static String valueOf(Object obj)	Produce la representación *string* de un *object*.

Creación de cadenas

"Entonces, Java incluye una clase *String* para gestionar las cadenas de texto", dice el programador novato. "Eso es fabuloso, porque estoy escribiendo esta novela, mire, y".... "Mantengámoslo", le dice. "No quiero saber nada más sobre eso".

Echemos un vistazo a algunas de las muchas formas de crear objetos *String*. Esta es una forma que ya hemos visto:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";
        .
        .
        .
```

De hecho, cuando en el código se usa un literal de cadena como "¡Hola desde Java!", Java lo trata como un objeto *String*; por tanto, lo que realmente está ocurriendo es que se está asignando un objeto *String* a otro.

Por supuesto, se puede declarar una cadena primero y luego asignarle un valor:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        String s2;
        s2 = "¡Hola desde Java!";

        .
        .
        .
```

Este es un caso en el que usamos uno de los constructores de la clase *String*. En este caso, se crea una cadena vacía y luego se le asignan datos:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        String s2;
        s2 = "¡Hola desde Java!";

        String s3 = new String( );
```

```
s3 = "¡Hola desde Java!";
.
.
.
```

Además al constructor de la clase *String* se le puede pasar, directamente, una cadena de texto para crear una nueva cadena, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        String s2;
        s2 = "¡Hola desde Java!";

        String s3 = new String();
        s3 = "¡Hola desde Java!";

        String s4 = new String("¡Hola desde Java!");
        .
        .
        .
```

Hay otros constructores de la clase *String* que pueden tomar *arrays* de caracteres o subconjuntos de *arrays* de caracteres (la clase *String* sabe qué constructor se está usando por el número y tipo de argumentos que se le pasan). Incluso se puede usar el método *valueOf* de la clase *String* para obtener una representación, en cadena, de valores numéricos:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        String s2;
        s2 = "¡Hola desde Java!";

        String s3 = new String();
        s3 = "¡Hola desde Java!";

        String s4 = new String("¡Hola desde Java!");

        Char c1[ ] = {'H', 'o', 'l', 'a', ' ', 'a', 'h', 'í'};
        String s5 = new String(c1);

        String s6 = new String(c1, 0, 4);

        double double1 = 1.23456789;
        String s7 = String.valueOf(double1);

        System.out.println(s1);
```

```
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(s4);
        System.out.println(s5);
        System.out.println(s6);
        System.out.println(s7);
    }
}
```



Truco: para convertir una cadena a número, se pueden usar las clases numéricas, *Integer*, *Long*, *Floar* y así sucesivamente, usando métodos como *Integer.parseInt* y *LongparseLong*.

Al final de este código, visualizamos todas las cadenas que se han creado. Esto es lo que aparecería al ejecutar el programa:

```
C:\>java app
¡Hola desde Java!
¡Hola desde Java!
¡Hola desde Java!
¡Hola desde Java!
Hola ahí
Hola
1.23456789
```

Obtención de la longitud de la cadena

El programador novato está sin aliento. "He escrito la mitad de mi novela", dice, "y necesito averiguar cuánto me queda. ¿Cómo puedo hacerlo?"

"Use el método *length* de la clase *String*", le dice.

Este es un ejemplo en el que se muestra cómo se usa el método *length* de la clase *String* (observe que además se puede ver que Java trata los literales cadena como objetos *String* usando *length* en un literal de cadena):

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        System.out.println("\"" + s1 + "\"" + " is " + s1.length()
            + " characters long".);
        System.out.println("\"" + "Hola" + "\"" + " tiene " +
            "Hola".length() + " caracteres".);
    }
}
```

Esta es la salida del programa:

```
C:\>java app
"¡Hola desde Java!" tiene 17 caracteres.
"Hola" tiene 4 caracteres.
```

Concatenación de cadenas

La concatenación de cadenas significa ponerlas juntas, y ya hemos usado el operador + en este libro para hacer esto. Sin embargo, hay otra forma de concatenar cadenas, se puede usar el método *concat* de la clase *String* para unir dos cadenas y crear una nueva. ¿Cómo se programa eso? Este es un ejemplo en el que se usa tanto el operador + como el método *concat* para crear la misma cadena:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola";

        String s2 = s1 + " desde";
        String s3 = s2 + " Java!";

        String s4 = s1.concat(" desde");
        String s5 = s4.concat(" Java!");
        System.out.println(s3);
        System.out.println(s5);
    }
}
```

Este es el resultado del código anterior:

```
C:\>java app
¡Hola desde Java!
¡Hola desde Java!
```

Como ya ha visto, a la hora de visualizar números, cuando se concatena un valor numérico con una cadena, el valor numérico se concatena como una cadena.



Truco: observe que al concatenar números hay que tratarlos como cadenas; por lo tanto, hay que tener cuidado. Por ejemplo, System.out.println("3 + 3 =" + 3 + 3) visualiza 3 + 3 = 33, no 3 + 3 = 6.

Obtención de caracteres y *substring*

La clase *String* proporciona un número de métodos que descompone las cadenas en sus componentes, caracteres y *substring*. Por ejemplo, se puede usar el método *charAt* para obtener el carácter que ocupa una posición determinada:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        char c1 = s1.charAt(0);
        System.out.println("El primer carácter de \"" + s1 +
            "\" es " + c1);
        .
        .
        .
```

Se puede usar el método *toCharArray* para convertir un objeto *String* en un *array* de caracteres, y se puede usar el método *getChars* para obtener un número de caracteres:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        char c1 = s1.charAt(0);
        System.out.println("El primer carácter de \"" + s1 +
            "\" es " + c1);

        char chars1[ ] = s1.toCharArray( );
        System.out.println("El segundo carácter de \"" +
              s1 + "\" es " + chars1[1];

        char chars2[ ] = new char[5];
        s1.getChars(0, 5, chars2, 0);
        System.out.println("Los primeros cinco caracteres de \"" + s1
              + "\" son " + new String(chars2));
        .
        .
        .
```

Además, puede usar el método *substring* para crear una cadena nueva que es un *substring* de la antigua, como sigue:

```
public class app
{
```

```
    public static void main(String[] args)
    {
        String s1 = "¡Hola desde Java!";

        char c1 = s1.charAt(0);
        System.out.println("El primer carácter de \"" + s1 +
            "\" es " + c1);

        char chars1[] = s1.toCharArray();
        System.out.println("El segundo carácter de \"" +
            s1 + "\" es " + chars1[1]);

        char chars2[] = new char[5];
        s1.getChars(0, 5, chars2, 0);
        System.out.println("Los primeros cinco caracteres de \"" + s1
            + "\" son " + new String(chars2));

        String s2 = s1.substring(0, 5);
        System.out.println("Los primeros cinco caracteres de \"" + s1
            + "\" son " + s2);
    }
}
```

Este es el resultado del programa:

```
C:\>java app
El primer carácter de "¡Hola desde Java!" es ¡
El segundo carácter de "¡Hola desde Java!" is H
Los primeros cinco caracteres de "¡Hola desde Java!" son ¡Hola
Los primeros cinco caracteres de "¡Hola desde Java!" son ¡Hola
```

Búsqueda y reemplazamientos en cadenas

Se pueden buscar caracteres y *substrings* utilizando los métodos *indexOf* y *lastIndexOf*. El método *indexOf* devuelve el lugar, tomando como base el cero, de la primera ocurrencia del carácter o *substring* en una cadena y *lastIndexOf* devuelve la localización de la última ocurrencia de un carácter o *substring*.

Esto es un ejemplo que prueba como se usa *indexOf* y *lastIndexOf*:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "He dibujado un bonito dibujo".;

        System.out.println("La primera vez que aparece \"dibu\" es " +
            "en la posición " + s1.indexOf("dibu"));

        System.out.println("La última vez que aparece \"dibu\" es " +
```

```
                "en la posición " + s1.lastIndexOf("dibu"));
        .
        .
        .
```

La clase *String* tiene también el método *replace*, que permite reemplazar todas las ocurrencias de un carácter simple por otro carácter. Quizás piense que esto va en contra de la idea de no poder cambiar el texto en un objeto *String*; sin embargo, este método crea un nuevo objeto *String*. Esto es un ejemplo que muestra cómo funciona (observe que se están cambiando todas las ocurrencias de la letra h por la letra f en una cadena de texto):

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "He dibujado un bonito dibujo".;

        System.out.println("La primera vez que aparece \"dibu\" es " +
            "en la posición " + s1.indexOf("dibu"));

        System.out.println("La última vez que aparece \"dibu\" es " +
            "en la posición " + s1.lastIndexOf("dibu"));

         String s2 = "Edna, you\'re hired!";
         System.out.println(s2.replace('h', 'f'));

    }
}
```

Este es el resultado del código:

```
C:\>java app
La primera vez que aparece "dibu" es en la posición 3
La última vez que aparece "dibu" es en la posición 22
Edna, you're fired!
```

Cambio de mayúsculas a minúsculas (o viceversa) en cadenas

El programador novato dice, "El gran jefe me dijo que la salida de mi programa no tenía énfasis suficiente. ¿Tiene alguna sugerencia?" "Intente usar el método *UpperCase*", le dice.

Se puede usar el método *toLowerCase* para convertir a minúsculas una cadena, y el *toUpperCase* para convertirla en mayúsculas. Así sería el código:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!".toLowerCase( ));
        System.out.println("¡Hola desde Java!".toUpperCase( ));
    }
}
```

Este es el resultado del programa:

```
C:\>java app
¡hola desde java!
¡HOLA DESDE JAVA!
```

Formateo de números en cadenas

Se pueden formatear números en cadenas utilizando la clase *NumberFormat* del paquete java.text. Esta clase soporta los métodos *format*, *set-Minimum-Integer-Digits*, *set-Minimum-Fraction-Digits*, *set-Maximum-integer-Digits* y *set-Maximum-Fraction-Digits*. Esto es un ejemplo, en el que se usa el método *set-Maximum-Fraction-Digits*, redondeando un valor *double* al formatearlo:

```
import java.text.*;
public class app
{
    public static void main(String[] args)
    {
        double value = 1.23456789;
        NumberFormat nf = NumberFormat.getNumberInstance();

        Nf.setMaximumFractionDigits(6);

        String s = nf.format(value);

        System.out.println(s);
    }
}
```

Este es el resultado:

```
C:\>java app
1.234568
```

La clase *StringBufferr*

"Hmm", dice el programador novato. "He almacenado toda mi novela en un objeto *String* y ahora no puedo cambiarlo. ¿Qué problema hay?" "No

puede cambiar el texto en el objeto *String*", le dice. "Tiene que usar un objeto *StringBuffer* en su lugar". "Entonces, cuénteme", contesta el NP.

La clase *StringBuffer* le da tanto y más como lo que ofrece la clase *String*: la posibilidad de modificar la cadena actual. Este es un ejemplo en el que se usa el método *replace* de la clase *StringBuffer* para cambiar el contenido de un objeto *StringBuffer* de "¡Hola desde Java!" a "¡Hola a Java!":

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer("¡Hola desde Java!");

        s1.replace(6, 11, "a");

        System.out.println(s1);
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola a Java!
```

Creación de *StringBuffers*

Se pueden crear objetos *StringBuffers* usando los constructores de la clase *StringBuffer*. Por ejemplo, aquí tenemos cómo se crea un objeto *StringBuffer* vacío (que se inicializa con 16 espacios en blanco, por defecto) y luego introducimos algo de texto:

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StrinBuffer( );
        s1.insert(0, "¡Hola desde Java!");

        System.out.println(s1);
        .
        .
        .
```

Tabla 2.5. Constructores de la clase *StringBuffer*.

Constructores	Descripción
StringBuffer( )	Construye un *buffer* de tipo *string*, sin caracteres y con una capacidad de 16 caracteres.

Constructores	Descripción
StringBuffer(int length)	Construye un *buffer* de tipo *string*, sin caracteres y con la capacidad marcada por el argumento *length*.
StringBuffer(String str)	Construye un *buffer* de tipo *string* que representa la misma secuencia de caracteres que el argumento.

Tabla 2.6. Métodos de la clase *StringBuffer*.

Método	Descripción
StringBuffer append(boolean b)	Añade al *buffer* la representación *string* del argumento booleano.
StringBuffer append(char c)	Añade al *buffer* la representación *string* del argumento *char*.
StringBuffer append(char[ ] str)	Añade al *buffer* la representación *string* del *array* de caracteres pasado como argumento.
StringBuffer append(char[ ] str, int offset, int len)	Añade la representación *string* del argumento subarray del *array* de caracteres al *buffer string*.
StringBuffer append(double d)	Añade al *buffer* la representación *string* del argumento *double*.
StringBuffer append(float f)	Añade al *buffer* la representación *string* del argumento *float*.
StringBuffer append(int i)	Añade al *buffer* la representación *string* del argumento *int*.
StringBuffer append(long l)	Añade al *buffer* la representación *string* del argumento *long*.
StringBuffer append(Object obj)	Añade al *buffer* la representación *string* del argumento *Object*.
StringBuffer append(String str)	Añade al *buffer* la cadena pasada como argumento.
Char charAt(int index)	Devuelve el carácter que ocupa la posición marcada por el argumento *index* en la secuencia representada por el *buffer*.

Método	Descripción
StringBuffer delete(int start, int end)	Borra los caracteres en un *substring* de este *buffer*.
StringBuffer deleteCharAt(int index)	Borra el caracter del *buffer* que ocupa la posicion dada por el argumento, acortando el *buffer* en un carácter.
void ensureCapacity(int minimum Capacity)	Asegura que la capacidad del *buffer* sea al menos igual al mínimo dado
void getChars(int srcBegin, intsrcEnd, char[ ] dst, int dstBegin)	Los caracteres se copian de este *buffer string* en el *array* de caracteres de destino.
StringBuffer insert(int offset, boolean b)	Inserta la representación *string* del argumento booleano en el buffer *string*.
StringBuffer insert(int offset, char c)	Inserta la representación *string* del argumento *char* en el *buffer*.
StringBuffer insert(int offset, char[ ] str)	Inserta la representación *string* del *array* de caracteres del argumento en el *buffer string*.
StringBuffer insert(index, char[ ] str, int offset, int len)	Inserta la representación de un sub-array del array str del argumento en el buffer string.
StringBuffer insert(int offset, double d)	Inserta la representación *string* del argumento *double* en el *buffer string*.
StringBuffer insert(int offset, float f)	Inserta en el *buffer* la representación *string* del argumento *float*.
StringBuffer insert(int offset, int i)	Inserta en el *buffer* la representación *string* del segundo argumento.
StringBuffer insert(int offset, long l)	Inserta en el *buffer* la representación del argumento *long*.
StringBuffer insert(int offset, Object obj)	Inserta la representación del argumento Object en el *buffer string*.
StringBuffer insert(int offset, String str)	Inserta la cadena en el *buffer string*.
int length( )	Devuelve la longitud (en caracteres) del *buffer*.

Método	Descripción
StringBuffer replace(start, int end, *String str)*	Reemplaza los caracteres del *substring* del *buffer* string con los caracteres del *string* dado.
StringBuffer reverse( )	La secuencia de caracteres del *buffer* es reemplazado por su inversa.
void setCharAt(int index, char ch)	El carácter del *buffer* que ocupa la posición dada por el índice se cambia por el marcado en el segundo argumento.
void setLenght (int newLength)	Establece la longitud del *string*.
String substring(int start)	Produce una nueva cadena que contiene una secuencia de caracteres contenida en el *buffer*. El *substring* empieza en el índice dado.
String substring(int start, int end)	Produce una nueva cadena que contiene una subsecuencia de caracteres actualmente contenida en el *buffer*.
String toString( )	Convierte los datos del *buffer* a un *string*.

Ahora, veamos cómo se inicializa un nuevo objeto *StringBuffer* con una cadena:

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer();
        s1.insert(0, "¡Hola desde Java!");
        System.out.println(s1);

        StringBuffer s2 = new StringBuffer("¡Hola desde Java!");
        System.out.println(s2);
        .
        .
        .
```

Además se puede crear un objeto *StringBuffer* con una longitud específica, como sigue:

```
public class app
{
```

```
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer();
        s1.insert(0, "¡Hola desde Java!");
        System.out.println(s1);

        StringBuffer s2 = new StringBuffer("¡Hola desde Java!");
        System.out.println(s2);

         StringBuffer s3 = new StringBuffer(10);
         s3.insert(0, "¡Hola desde Java!");
         System.out.println(s3);
    }
}
```

Este es el resultado del programa:

```
C:\>java app
¡Hola desde Java!
¡Hola desde Java!
¡Hola desde Java!
```

Obtención y establecimiento de longitudes y capacidades de *StringBuffer*

Se puede usar el método *length* de la clase *StringBuffer* para buscar las longitudes de los objetos en *StringBuffer*, y se puede usar el método *capacity* para localizar la cantidad de espacio de memoria alocado para ese texto. Además, se puede fijar la longitud del texto en un objeto *StringBuffer* con el método *setLength*, que permite truncar cadenas o extenderlas con caracteres nulos (es decir, caracteres cuyos códigos Unicode son 0).

Esto es un ejemplo que muestra cómo determinar la longitud de una cadena, cómo determinar su capacidad (Java normalmente hace que la capacidad sea 16 caracteres mayor que la longitud, para ahorrar tiempo para futuras alocaciones de memoria), y cómo cambiar la longitud de la cadena:

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer("¡Hola desde Java!");

         System.out.println("La longitud es " + s1.length( ));
         System.out.println("La longitud reservada es " +
                  s1.capacity( ));

         s1.setLength(2000);
```

```
            System.out.println("La nueva longitud es " + s1.length( ));
        }
}
```

Esto es lo que el programa muestra al ejecutarlo:

```
C:\>java app
La longitud es 17
La longitud alocada es 33
La nueva longitud es 2000
```

Establecer caracteres en *StringBuffers*

"¡Socorro!" grita el programador novato, "¡necesito cambiar algo de texto en mi novela!" "Puede intentarlo con el método *setCharAt*", le dice.

Para leer caracteres de un objeto *StringBuffer*, se pueden usar los métodos *charAt* y *getChars*, al igual que con los objetos *String*. Sin embargo, en los objetos *StringBuffer*, también se pueden introducir caracteres individuales usando el método *setCharAt*.

Esto es un ejemplo en el que se cambia el texto "She had a wild look in her eyes". por "She had a mild look in her eyes". usando *setCharAt*:

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new
            StringBuffer("She had a wild look in her eyes".);

         s1.setCharAt(10, 'm');

        System.out.println(s1);
    }
}
```

Este es el resultado:

```
C:\>java app
She had a mild look in her eyes.
```

Añadir e insertar utilizando *StringBuffers*

"El método *setCharAt* no lo hace", dice el programador novato. "Necesito alguna forma de editar el texto de los objetos *StringBuffer* como una cadena, no como caracteres individuales". "De acuerdo", le dice, "use los métodos *append* e *insert*".

Se puede usar el método *append* para añadir cadenas al texto en un objeto *StringBuffer*, y el método *insert* para insertar texto en un lugar en particular. Este es un ejemplo que empieza con "¡Hola"; se le añade "Java!", y luego se le inserta "desde" en el medio del texto, usando *append* e *insert*:

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer("¡Hola");

        s1.append(" Java!");

        System.out.println(s1);

        S1.insert(6, "desde ");

        System.out.println(s1);
    }
}
```

Esto es lo que produce el código:

```
C:\>java app
¡Hola Java!
¡Hola desde Java!
```

Borrar texto en *StringBuffers*

Se puede borrar texto en un objeto *StringBuffer* usando los métodos *delete* y *deleteCharAt*.

Por ejemplo, aquí podemos ver cómo se cambia el texto "No tengo un buen momento" por "Tengo un buen momento" con *delete* (al usar este método, se especifica el rango de caracteres que se quieren borrar):

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new
            StringBuffer("No tengo un buen momento");

        s.delete(0, 2);
        System.out.println(s1);
    }
}
```

Este es el resultado:

```
C:\>java app
tengo un buen momento.
```

Reemplazar texto en *StringBuffers*

"Estoy escribiendo un editor de texto usando la clase *StringBuffer*", dice el programador novato, "pero hay una cosa que no entiendo, ¿cómo puedo reemplazar un texto por otro? ¿Tengo que borrarlo y luego insertar el nuevo?" "No", le contesta. "Es fácil, basta con usar el método *replace*".

De hecho, ya se ha visto cómo se usa *replace*; sólo hay que especificar un rango de caracteres y el nuevo texto que debería sustituir al rango marcado, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer("¡Hola desde Java!");

        s.replace(6, 11, "a");

        System.out.println(s1);
    }
}
```

Este es el resultado del código anterior:

```
C:\>java app
¡Hola a Java!
```

3 Operadores, condicionales y bucles

En el capítulo anterior, vimos cómo Java gestiona los datos desde un punto de vista básico. En este capítulo, empezaremos a trabajar con esos datos, como examinar los operadores, condicionales y bucles en Java.

Almacenar muchos datos en el programa es bueno, siempre y cuando se haga algo con ellos. Utilizando operadores, se pueden manipular los datos, sumar, restar, multiplicar, dividir y mucho más. Con los condicionales, se puede alterar el flujo de un programa evaluando los valores de los datos. Con los bucles, se puede iterar sobre todos los datos de un grupo, como un *array*, trabajando con ellos sucesivamente de forma fácil. Estos tres casos son el siguiente paso en la potencia de la programación y los discutiremos en este capítulo.

Operadores

La forma más básica de trabajar con los datos en un programa es hacerlo con los operadores de Java. Por ejemplo, supongamos que se ha almacenado un valor de 46 en una variable, y el valor 4, en otra. Se pueden multiplicar esos dos valores con el operador de multiplicación (*), como se indica en este código:

```
public class app
{
    public static void main(String[] args)
    {
        int operand1 = 46, operand2 = 4, product;

            product = operand1 * operand2;

        System.out.println(operand1 + " * " + operand2 +
            " = " + product);
    }
}
```

Este es el resultado del código:

```
C:\>java app
46 * 4 = 184
```

¿Qué operadores hay en Java? Esta es la lista:

- -- (decremento)
- - (resta)
- ! (No lógico)
- != (distinto)
- % (módulo)
- %= (asignación del módulo)
- & (AND a nivel de bit)
- && (AND en cortocircuito)
- &= (asignación de AND)
- * (multiplicación)
- *= (asignación del producto)
- / (división)
- /= (asignación de la división)
- ?: (*if-then-else*)
- ^ (Xor a nivel de bit)
- ^= (asignación de Xor)
- | (OR a nivel de bit)
- || (OR en cortocircuito)

- |= (asignación de OR)
- ~ (NOT unario a nivel de bit)
- + (suma)
- ++ (incremento)
- += (asignación de la suma)
- < (menor que)
- << (desplazamiento de bits hacia la izquierda)
- <<= (asignación del desplazamiento de bits a la izquierda)
- <= (menor o igual que)
- = (asignación)
- -= (asignación de la resta)
- == (igual a)
- > (mayor que)
- >= (mayor o igual que)
- >> (desplazamiento a la derecha)
- >>= (asignación del desplazamiento a la derecha)
- >>> (desplazamiento a la derecha con relleno de ceros)
- >>>= (asignación del desplazamiento a la derecha con relleno de ceros)

Se verán todos estos operadores a lo largo de este capítulo. Aquellos que tienen sólo un operando se llaman operadores unarios. Los que tienen dos operandos, por ejemplo, la suma (a + b), se llaman binarios. Además, hay un operador, ?:, que tiene tres operandos: el operador ternario.



Nota: a lo largo del capítulo, además de estos operadores, veremos la clase *Math* de Java, que permite añadir más capacidad matemática a los programas, incluyendo la potenciación (a diferencia de otros lenguajes, Java no tiene un operador de potenciación), logaritmos, funciones trigonométricas y mucho más.

Condicionales

Después de los operadores, el siguiente paso es usar las sentencias condicionales, también llamadas instrucciones de control de flujo. Se utilizan para

tomar decisiones basadas en el valor de los datos y dirigir el flujo del programa de acuerdo a esos valores.

Porque, supongamos que queremos informar sobre el tiempo y si está por debajo de 80 grados Fahrenheit, visualizar el mensaje "No hace demasiado calor.". Esto se puede hacer comprobando la temperatura actual con una sentencia *if* de Java que compara el valor de la variable *temperature* con 80 y que, si el valor es menor que 80, visualiza el mensaje:

```
public class app
{
    public static void main(String[] args)
    {
        int temperature = 73;

        if (temperature < 80) {
              System.out.println("No hace demasiado calor.");
        }
    }
}
```

La sentencia *if* comprueba si su condición (la parte que aparece entre paréntesis) es verdadera, que en este caso es *temperature* < 80. El operador relacional < (menor que) se usa para ver si el valor de la variable *temperature* es menor que 80. Dado que esta variable se ha inicializado a 73, la condición de la sentencia *if* es verdadera, lo que significa que se va a ejecutar el bloque de código de la sentencia *if*. Este es el resultado:

```
C:\>java app
No hace demasiado calor.
```

Las sentencias *if* pueden ser más complejas si se añaden las cláusulas *else*. Deben seguir a la sentencia *if* y se ejecutan cuando su condición es falsa. Un ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
        int temperature = 73;

        if (temperature < 80) {
            System.out.println("No hace demasiado calor.");
        }
         else {
            System.out.println("¡Hace mucho calor!");
         }
    }
}
```

Como veremos en este capítulo, hay otras sentencias condicionales.

Bucles

Los bucles son fundamentales en la programación y permiten gestionar distintas tareas repitiendo la ejecución de cierto código específico. Por ejemplo, puede que quiera gestionar los elementos de un grupo de datos trabajando con cada uno de ellos en serie, o quizás quiera ejecutar una tarea hasta que cierta condición sea verdadera. El bucle básico involucra a la sentencia *for*, que permite ejecutar un bloque de código usando un índice. Cada vez que se pasa por el bucle, el índice tendrá un valor diferente y se puede usar para hacer referencia a cada elemento del conjunto de datos. El índice del bucle se usa como si fuera un índice de un *array*.

A continuación se muestra el formato general del bucle *for* (observe que la sentencia que forma parte del cuerpo del bucle *for* puede ser compuesta, lo que significa que pueden presentarse varias sentencias sencillas encerradas entre paréntesis):

```
for (expresión_de_inicialización; condición_final; expresión_de_iteración)
{
    sentencia
}
```

Se puede inicializar un índice en expresión_de_inicialización (de hecho, se pueden usar varios índices en un bucle *for*), proporcionar una condición para finalizar el bucle en condición_final y añadir alguna forma de cambiar (normalmente incrementando) el índice en expresión_de_iteración.

Para aclarar todo esto, pongamos un ejemplo. En este caso, usaremos el bucle *for* para acumular los grados de seis estudiantes de un *array* y calcular el grado medio. Así aparece el código (observe que, realmente, se está declarando e inicializando el índice del bucle a 0 en la expresión de inicialización del bucle *for*, ya que Java lo permite, al igual que C++):

```
public class app
{
    public static void main(String[] args)
    {
        double grades[] = {88, 99, 73, 56, 87, 64};
        double sum, average;

        sum = 0;

      for (int loop_index = 0; loop_index < grades.length; loop_index++)
        {
              sum += grades[loop_index];
        }

        average = sum / grades.length;
```

```
        System.out.println("Grado medio = " + average);
    }
}
```

Este código recorre todos los grados del *array* y los va sumando, dejando el resultado en la variable *sum*, que después se divide entre el número total de elementos del *array* para calcular el grado medio. Todos los elementos se recorren utilizando un índice que empieza en 0 y en cada paso se incrementa hasta llegar al último elemento del *array*. Este es el resultado del código:

```
C:\>java app
Grado medio = 77.83333333333333
```

Como se puede ver, el bucle *for* es potente; de hecho es uno de los muchos puntos que se tratarán en las secciones siguientes. Ya es hora de empezar a usar los operadores, las sentencias condicionales y los bucles.

Precedencia de operadores

"Oye", dice el programador novato, "Java está actuando mal otra vez. Estaba tratando de sumar 12 más 24 y luego dividir el resultado entre 6. La respuesta debería ser 6 y Java devuelve 16". "Probablemente es un problema de precedencia de los operadores", le respondemos. "Revisemos el código".

Java soporta gran cantidad de operadores y puede haber problemas si se utilizan muchos en una sentencia simple. ¿Qué operador se ejecuta primero? Por ejemplo, veamos el código del programador novato, en el que intenta sumar 12 más 24 y dividir la suma entre 6:

```
public class app
{
    public static void main(String[] args)
    {
        double value;

        value = 12 + 24 / 6;

        System.out.println("El valor = " + value);
    }
}
```

Este es el resultado del código:

```
C:\>java app
El valor = 16.0
```

Claramente, hay algo que se ha hecho de distinta forma a la que el programador novato esperaba. De hecho, Java tiene una precedencia de operado-

res muy clara, lo que quiere decir que si encuentra dos operadores del mismo nivel en una sentencia (es decir, no encerrada entre paréntesis), primero ejecutará el operador de mayor precedencia. Esto es lo que ocurre aquí; que el operador / tiene mayor precedencia que el operador +, por lo tanto en la expresión anterior, primero se divide 24 entre 6 y el resultado se divide entre 12 dando 16.

Para especificar en Java el orden que se quiere, se puede usar paréntesis para agrupar las operaciones que se quieren ejecutar primero. Así quedaría el ejemplo anterior, en el que se ha puesto entre paréntesis "12 + 24", para asegurar que esta operación se ejecuta primero:

```
public class app
{
    public static void main(String[] args)
    {
        double value;

        value = (12 + 24) / 6;

        System.out.println("El valor = " + value);
    }
}
```

Este es el resultado del código:

```
C:\>java app
El valor = 6.0
```

La tabla 3.1 explica la precedencia de operadores en Java (desde la más alta a la más baja). Observe que en el nivel de precedencia más alto, encontrará ( ), [ ] (el "operador" *array*, que se usa para obtener los datos de un índice específico en un *array*), y . (el operador punto, que se usa para especificar los métodos y los miembros de datos de los objetos). Esto significa, por ejemplo, que siempre se pueden usar paréntesis para establecer el orden de ejecución de las operaciones en las expresiones de Java.

En este capítulo, veremos todos estos operadores de Java en orden de precedencia. Empezaremos por los operadores incremento y decremento: ++ y --.

Incremento y decremento: ++ y --

El zar de la Programación Exacta aparece y dice, "C++ tiene un operador de incremento y otro de decremento. ¿Java los soporta?" "Claro", le dice.

El operador de incremento ++ suma uno al operando y el operador de decremento -- se lo resta. Por ejemplo, si *value* tiene el valor 0, después de ejecutar *value*++, tendrá el valor 1. Estos operadores se introdujeron en C para incrementar y decrementar valores fácilmente, por ser operaciones muy comunes. De hecho, fueron tan populares que el operador de incremento se usó en el nombre de C++, indicando que C++ es una versión incremental de C.

Tabla 3.1. Precedencia de operadores.

Operadores		
( )	[ ]	.
++	--	~ !
*	/	%
+	-	
>>	>>>	<<
>	>=	< <=
==	!=	
&		
^		
|		
&&		
||		
?:		
=	[operador]=	

Aquí hay un punto importante: ++ y -- pueden ser operadores sufijo (por ejemplo, *value*++) o prefijo (++*value*). Cuando se usan como operadores sufijo, se ejecutan después del resto de la expresión y cuando se usan como operadores prefijo, se ejecutan antes que el resto de la expresión. Esto es algo que hay que comprobar. Por ejemplo, veamos el siguiente código:

```
value2 = value1++;
```

En este caso, cuando la sentencia se completa, *value2* contendrá el valor original de *value1* y el valor de *value1* se incrementará. He quí un ejemplo de cómo funciona:

```
public class app
{
    public static void main(String[] args)
    {
        int value1 = 0, value2 = 0;

        System.out.println("valor1 = " + value1);
        System.out.println("valor2 = " + value2);


        value2 = value1++;

       System.out.println("Después de ejecutar value2 = ++value1...");
        System.out.println("valor1 = " + value1);
        System.out.println("valor2 = " + value2);

        int value3 = 0, value4 = 0;

        System.out.println();
        System.out.println("valor3 = " + value3);
        System.out.println("valor4 = " + value4);

        value4 = ++value3;

       System.out.println("Después de ejecutar value4 = ++value3...");
        System.out.println("valor3 = " + value3);
        System.out.println("valor4 = " + value4);
    }
}
```

Estos son los resultados:

```
C:\>java app
valor1 = 0
valor2 = 0
Después de ejecutar value2 = ++value1...
valor1 = 1
valor2 = 0

valor3 = 0
valor4 = 0
Después de ejecutar value4 = ++value3...
valor3 = 1
valor4 = 1
```

NOT unario: ~ y !

El operador ~ es el operador unario NOT a nivel de bit y ! es el operador unario lógico NOT. El operador ~ cambia todos los bits de los argumentos numéricos y el operador ! cambia los valores verdaderos a falsos y viceversa.

Esto es un ejemplo en el que se cambian todos los bits del mayor valor positivo que puede ser almacenado en un dato de tipo *short*, para obtener el valor más negativo posible de un *short*, y además se cambia un valor booleano de verdadero a falso:

```
public class app
{
    public static void main(String[] args)
    {
        short short1 = 32767;
        boolean boolean1 = true;

System.out.println("El valor más negativo de un short = " + ~short1);
        System.out.println("!verdadero =" + !boolean1);

    }
}
```

Este es el resultado:

```
C:\>java app
El valor más negativo de un short = -32768
!verdadero = false
```

Si se inicializa *int1* a 0 y luego se cambian sus bits con el operador ~ a 1111111111111111 en binario, Java habría visualizado el resultado -1, porque usa la notación de complemento a dos para los números negativos. Esto significa que el bit que ocupa el primer lugar en los números negativos es 1 y 0 para los positivos.



Truco: ¿Por qué 1111111111111111 binario es igual a -1 en una variable de tipo *short*? Si se le sumara a 1, se tendría 10000000000000000 en binario, un número demasiado grande para los 16 bits de un *short*, por lo que el 1 del primer lugar se pierde y se obtiene 0, en otras palabras, -1 + 1 = 0.

Multiplicación y división: * y /

El zar de la Programación Exacta dice, "Espero que Java tenga operadores de multiplicación y división, como C++". "Claro, " le dice.

En Java, se utiliza * para multiplicar y / para dividir valores. Veamos un ejemplo en el que se usa * y / con valores *double*, y luego se hace lo mismo con valores enteros. Se ejecuta la multiplicación y división con valores enteros para mostrar que, cuando se usan enteros, la parte fraccionaria de los

resultados matemáticos se truncan. Por lo tanto, si se quiere ejecutar una operación de división y mantener la precisión, probablemente no se deberían usar enteros. Este es el código:

```
public class app
{
    public static void main(String[] args)
    {
        double double1 = 4, double2 = 6, double3 = 5, doubleResult;

        doubleResult = double1 * double2 / double3;

        System.out.println("4 * 6 / 5 = " + doubleResult);


        int int1 = 4, int2 = 6, int3 = 5, intResult;

        intResult = int1 * int2 / int3;

        System.out.println("Con enteros, 4 * 6 / 5 = " + intResult);
    }
}
```

Este es el resultado:

```
C:\>java app
4 * 6 / 5 = 4.8
Con enteros, 4 * 6 / 5 = 4
```

Módulo: %

Se usa el operador módulo (%) para devolver el resto de una operación de división. Por ejemplo, 10/3 es igual a 3 con un resto de 1. Observe que el operador módulo es especialmente útil cuando se hace conversión entre bases, porque se puede usar con la base a la que se está convirtiendo. Para ver cómo funciona, eche un vistazo a la sección "Bucle *for*", más adelante. En ella hay un ejemplo completo.

Suma y resta: + y -

"El operador multiplicación es un asterisco y el de la división es una barra inclinada", dice el programador novato, "pero esos no son los signos que se aprenden en el colegio. ¿Qué utiliza Java para los signos más y menos?" "Los símbolos usuales", le contestamos.

Los operadores numéricos más antiguos son + y -, que se usan para la suma y la resta, respectivamente. Por ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
        int operand1 = 5, operand2 = 4, sum, diff;

        sum = operand1 + operand2;
        diff = operand1 - operand2;

        System.out.println(operand1 + " + " + operand2 + " = " + sum);
        System.out.println(operand1 + " - " + operand2 + " = " + diff);
    }
}
```

Estos son los resultados:

```
C:\>java app
5 + 4 = 9
5 - 4 = 1
```

Operadores de desplazamiento: >>, >>> y <<

Se usan los operadores de desplazamiento para desplazar, hacia la izquierda o derecha del operando situado en el lado izquierdo, el número de bits indicado por el operando situado en el lado derecho. Hay tres operadores de desplazamiento: hacia la derecha (>>), hacia la derecha sin signo (>>>) y hacia la izquierda (<<). Así se usan estos operadores:

```
nuevo_valor = valor << número_de_lugares;
nuevo_valor = valor >> número_de_lugares;
nuevo_valor = valor >>> número_de_lugares;
```

Por ejemplo, 16 >> 2 desplaza 2 bits hacia la derecha del número 16, que es lo mismo que dividir entre 4; por lo tanto, 16 >> 2 es igual a 4. Normalmente, se usan los operadores de desplazamiento para empaquetar valores binarios en un *int* o *long* como campos, porque se puede sumar un número al *int* o *long* y luego desplazarlo hacia la izquierda para hacerle sitio al siguiente campo de datos.

Es importante que sepa que el operador >> respeta el signo de su operando y dado que un valor negativo significa que el bit de más a la izquierda es 1, el desplazamiento de un número negativo hacia la derecha introduce un nuevo 1 en su izquierda. Así, cambiar 1111111111111100, que es –4 como un *short*, devuelve 1111111111111110, que es –2. Además, cambiar –1, que es

1111111111111111, da 1111111111111111, que continúa siendo –1. Si realmente quiere trabajar con los bits actuales de un número cuando se los desplaza a la derecha y no tiene uno añadido a la izquierda cuando se desplazan a la izquierda números negativos, use el operador de desplazamiento sin signo hacia la derecha (>>>). Éste introduce un cero a la izquierda, si el número es positivo o negativo.

Veamos un ejemplo donde interviene el operador de desplazamiento:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 16, negValue = -1;

        System.out.println(value + " << 2 = " + (value << 2));
        System.out.println(value + " >> 2 = " + (value >> 2));
        System.out.println(negValue + " >> 2 = " + (negValue >> 2));
       System.out.println(negValue + " >>> 22 = " + (negValue >>> 22));
    }
}
```

Este es el resultado:

```
16 >> 2 = 4
-1 >> 2 = -1
-1 >>> 22 = 1023
```

Operadores de relación: >, >=, <, <=, == y !=

El Gran Jefe (GJ) aparece y dice, "Casi hemos gastado el presupuesto y hay que asegurarse de que no lo sobrepasamos". "Hmm", le dice, "parece que es un trabajo para el operador de relación *menor que*. Por cierto, sobre lo de mi ascenso...". "Olvídelo", dice el GJ.

Use los operadores relacionales para crear condiciones lógicas que puedan ser evaluadas con sentencias condicionales como *if*. Por ejemplo, veamos lo que deberíamos evaluar para asegurarnos que el presupuesto es mayor que cero utilizando la sentencia *if*:

```
public class app
{
    public static void main(String[] args)
    {
        int budget = 1;
        if (budget < 0) {
            System.out.println("Uh oh.");
        }
        else {
```

```
                System.out.println("Todavía es solvente.");
            }
        }
}
```

Este es el resultado:

```
C:\>java app
Todavía es solvente.
```

A continuación se muestra una lista de los operadores relacionales; estos operadores devolverán verdadero si sus operandos cumplen las descripciones dadas:

- > (mayor que; por ejemplo, operando1 > operando2 devuelve verdadero si el operando1 es mayor que el operando 2)
- >= (mayor o igual que)
- < (menor que)
- <= (menor o igual que)
- == (igual a)
- != (distinto)

Los operadores relacionales se pueden combinar con operadores lógicos (para más detalles, ver el siguiente punto).



Truco: a continuación veremos una trampa de Java, que hay que evitar: cuando se crea una condición lógica, hay que tener presente que probablemente se quiera usar == en lugar de =. Por ejemplo, la expresión *budget* == 0 es verdadera si el valor de *budget* es 0, pero la expresión *budget* = 0 asigna el valor 0 a *budget*. Tenga cuidado, porque usar = en lugar de == en condiciones lógicas es un error muy común.

Operadores lógicos a nivel de bit AND, Xor y OR: &, ^ y |

"¡Socorro!" dice el programador novato, "necesito saber si el bit número 3 de un entero está puesto a 1. ¿Hay alguna forma fácil de hacerlo?" "Claro", le dice, "puede usar un operador a nivel de bit".

Los operadores a nivel de bit permiten examinar cada uno de los bits de los valores. Por ejemplo, cuando se usa el operador a nivel de bit & con dos

operandos, se realiza la operación lógica AND con cada bit de un operando y su correspondiente en el otro. Si ambos bits son 1, un uno aparece en ese lugar en el resultado; de lo contrario, será un cero. Por ejemplo, se puede hacer la prueba del programador novato de si el tercer bit de un valor está a 1 haciendo el AND lógico con un número para el que se sabe que el tercer bit es 1. Si el resultado de la operación AND no es cero, el tercer bit del valor original era 1. Aquí se muestra cómo sería el código:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 12;
        int bit3setting = value & 1 << 3;

        if (bit3setting != 0) {
            System.out.println("Bit 3 está activo.");
        }
        else {
            System.out.println("Bit 3 no está activo.");
        }
    }
}
```

Este es el resultado:

```
C:\>java app
Bit 3 está activo.
```

Los operadores a nivel de bit se pueden encontrar en la tabla 3.2. En pocas palabras, así es cómo funcionan: El operador OR (|) devuelve 0 cuando ambos bits son 0 y devuelve 1 en caso contrario. El operador AND (&) devuelve 1 cuando ambos bits son 1 y devuelve 0 en caso contrario. Finalmente, el operador Xor (^, llamado OR exclusivo) devuelve 1 cuando un bit es 0 y el otro es 1 y devuelve 0 en caso contrario.

Cuando los operadores &, ^ y | operan con valores booleanos verdadero/falso, se consideran operadores lógicos a nivel de bit. Los operadores lógicos a nivel de bit funcionan igual que los operadores a nivel de bit (sustituyen falso por 0 y verdadero por 1), como se puede ver en la tabla 3.3.

En pocas palabras, así funcionan los operadores lógicos a nivel de bit: el operador OR (|) devuelve falso cuando ambos operandos son falsos y devuelve verdadero en caso contrario. El operador AND (&) devuelve verdadero cuando ambos operandos son verdaderos y devuelve falso en caso contrario. El operador Xor (^) devuelve verdadero cuando un operando es falso y uno es verdadero, y devuelve falso en caso contrario.

Tabla 3.2. Los operadores a nivel de bit.

x	y	x | y (O)	x & y (Y)	x ^ y (Xor)
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

Tabla 3.3. Los operadores lógicos a nivel de bit.

x	y	x | y (O)	x & y (Y)	x ^ y (Xor)
Falso	Falso	Falso	Falso	Falso
Verdadero	Falso	Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero	Falso	Verdadero
Verdadero	Verdadero	Verdadero	Verdadero	falso

Este es un ejemplo en el que se han puesto dos condiciones lógicas juntas, visualizando un mensaje en el caso de que cualquiera de ellas sea verdad, usando el operador |:

```
public class app
{
    public static void main(String[] args)
    {
        int budget = 1;
        boolean fired = false;

        if (budget < 0 | fired == true) {

            System.out.println("Uh oh.");
        }
        else {
            System.out.println("Todavía es solvente.");
        }
    }
}
```

Este es el resultado:

```
C:\>java app
Todavía es solvente.
```

En el siguiente ejemplo, insisto en que la temperatura está entre 60 y 90 grados, usando el operador lógico a nivel de bit &, antes de visualizar un mensaje:

```
public class app
{
    public static void main(String[] args)
    {
        int temperature = 70;

        if (temperature < 90 & temperature > 60) {
            System.out.println("Podemos ir de merienda.");
        }
    }
}
```

Este es el resultado:

```
C:\>java app
Podemos ir de merienda.
```

Como se puede ver, los operadores lógicos a nivel de bit pueden ser muy útiles. Java además incluye dos operadores lógicos: && y ||. Veámoslos a continuación.

&& y || lógicos

Los dos operadores lógicos que generalmente se usan en expresiones son AND (&&) y OR (||). La tabla 3.4 muestra cómo funcionan estos operadores.

El operador OR (||) devuelve falso cuando ambos operandos son falsos y devuelve verdadero en caso contrario. El operador AND (&&) devuelve verdadero cuando ambos operandos son verdaderos y devuelve falso en caso contrario. Estos operadores se usan para enlazar cláusulas lógicas, se usa AND cuando se quiere que ambas cláusulas sean verdad y OR cuando sólo se necesita que una de las dos sea verdadera.

Este es un ejemplo del punto anterior, en el que se usa &&:

```
public class app
{
    public static void main(String[] args)
    {
        int temperature = 70;

        if (temperature < 90 && temperature > 60) {
            System.out.println("Podemos ir de merienda.");
        }
    }
}
```

El resultado es:

```
C:\>java app
Podemos ir de merienda.
```

Los operadores && y || tienen además otra propiedad interesante: son operadores cortocircuito, lo que quiere decir que si pueden determinar todo lo que necesitan saber evaluando el operando de la izquierda, no evaluarán el operando derecho. Esto es muy útil en casos como el que sigue, en el que estamos comprobando si un valor tiene 0 y si u inverso es menor que 1000. Si el valor es 0, la segunda parte de la expresión, en la que se calcula su inverso, no se ejecuta. De esta forma, se evita un error de desbordamiento al intentar dividir entre cero.

Tabla 3.4. Los operadores lógicos.

x	y	x || y (O)	x && y (Y)
Falso	Falso	Falso	Falso
Verdadero	Falso	Verdadero	Falso
Falso	Verdadero	Verdadero	Falso
Verdadero	Verdadero	Verdadero	Verdadero

Este es el código:

```
public class app
{
    public static void main(String[] args)
    {
        double value = 0;

        if (value != 0 && 1 / value < 1000) {
            System.out.println("El valor no es demasiado pequeño.");
        }
        else {
            System.out.println("El valor es demasiado pequeño.");
        }
    }
}
```

Este es el resultado:

```
C:\>java app
El valor es demasiado pequeño.
```

Los operadores lógicos se diferencian de los operadores lógicos a nivel de bit en que los primeros son operadores cortocircuito. Para ver cómo funcionan, echemos un vistazo al código siguiente, en el que la asignación de la sentencia *if* se ejecuta cuando se usa el operador & pero no cuando se usa el operador cortocircuito &&:

```
public class app
{
    public static void main(String[] args)
    {
        double int1 = 0, int2 = 1, int3 = 1;

        if (int1 != 0 & (int2 = 2) == 1) { }
        System.out.println("int2 = " + int2);

        if (int1 != 0 & (int3 = 2) == 1) { }
        System.out.println("int3 = " + int3);
    }
}
```

Este es el resultado:

```
C:\>java app
int2 = 2.0
int3 = 1.0
```

El operador *If-Then-Else*: ?:

"De acuerdo", dice el programador novato (PN), "soy un experto en operadores. Estoy preparado para las sentencias condicionales de Java". "No tan rápido", le dice. "¿Qué hay sobre el operador condicional ternario?" "¿El qué?" pregunta PN.

Hay un operador Java que funciona como una sentencia *if-else*, el operador ternario (?:). Este operador se llama operador ternario porque involucra tres operandos, una condición y dos valores:

```
valor = condición ? valor1 : valor2;
```

Si la condición es verdadera, el operador ?: devuelve valor1 y, en caso contrario devuelve valor2. De esta forma, la sentencia precedente funciona como la siguiente sentencia *if*:

```
if (condición) {
    value = valor1;
}
else {
    value = valor2;
}
```

Veamos un ejemplo en el que un entero entre 0 y 15 se convierte a hexadecimal usando el operador ?:. Este operador es perfecto para esto, porque se puede usar para devolver una cadena construida a partir de un valor, si el valor es menor que 10 o un dígito si el valor es mayor o igual que 10, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 15;
        String digit, chars[] = {"a", "b", "c", "d", "e", "f"};

        Digit = value < 10 ? String valueOf(value) : chars[value - 10];

        System.out.println(value + " = 0x" + digit);
    }
}
```

Este es el resultado:

```
C:\>java app
15 = 0xf
```

Operadores de asignación: = y [operador]=

La mayor parte de los operadores básicos son operadores de asignación (ya hemos usado estos operadores en el libro). El operador = se puede utilizar para asignar a una variable un valor literal o el valor de otra variable, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 12;

        System.out.println("El valor = " + value);
    }
}
```

Este es el resultado:

```
C:\>java app
El valor = 12
```

Como en C++, se pueden ejecutar múltiples asignaciones en la misma sentencia (esto funciona porque el operador asignación, devuelve el valor asignado):

```
public class app
{
    public static void main(String[] args)
    {
        int value1, value2, value3;

        value1 = value2 = value3 = 12;
```

```
        System.out.println("valor1 = " + value1);
        System.out.println("valor2 = " + value2);
        System.out.println("valor3 = " + value3);
    }
}
```

Este es el resultado:

```
C:\>java app
valor1 = 12
valor2 = 12
valor3 = 12
```

Además, como en C++, se pueden combinar muchos operadores con el operador de asignación (=). Por ejemplo, += es el operador de asignación de la suma, que significa que *value* += 2 es una forma corta de *value* = *value* + 2. Esto es un ejemplo que utiliza el operador de asignación de la multiplicación:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 10;

        value *= 2;

        System.out.println("value * 2 = " + value);
    }
}
```

Este es el resultado:

```
C:\>java app
value * 2 = 20
```

Hay bastantes combinaciones de operadores de asignación. Esta es la lista:

- %= (asignación de módulo)
- &= (asignación a nivel de bit de AND)
- *= (asignación de multiplicación)
- /= (asignación de división)
- ^= (asignación a nivel de bit de Xor)
- |= (asignación a nivel de bit de OR)
- += (asignación de suma)
- <<= (asignación de desplazamiento a la izquierda)

- <= (menor o igual que)
- -= (asignación de resta)
- >>= (asignación de desplazamiento a la derecha)
- >>>= (asignación de desplazamiento a la derecha con relleno de ceros)

Esto completa la lista de los operadores de Java, pero hay una forma más popular de gestionar matemáticas en Java, la clase *Math*. Esta clase es parte del paquete java.lang (que el compilador Java importa por defecto). Echaremos un vistazo a esta clase en la siguiente sección.

Utilización de la clase *Math*

"Oye", dice el programador novato (PN)",quiero elevar 3 a la potencia 4, pero no hay ningún operador de Java para la potenciación". "Se puede usar el método *pow* de la clase *Math*", le dice. "Y de esta forma, 3 elevado a la potencia 4 es 81". "La crearé cuando Java me lo pida, " dice PN.

Se puede usar la clase java.lang.Math para ejecutar muchas operaciones matemáticas. Por ejemplo, así se resuelve el problema del programador novato usando el método *Math.pow*:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("3 x 3 x 3 x 3 = " + Math.pow(3, 4));
    }
}
```

Este es el resultado:

```
C:\>java app
3 x 3 x 3 x 3 = 81.0
```

Las constantes y métodos de la clase son:

- *double* E: El número 'e' (2.7182818284590452354)
- *double* PI: La constante pi (3.14159265358979323846)
- *double* sin(*double* a): Seno
- *double* cos(*double* a): Coseno

- *double* tan(*double* a): Tangente
- *double* asin(*double* a): Arcoseno
- *double* acos(*double* a): Arcocoseno
- *double* atan(*double* a): Arcotangente
- *double* atan2(*double* a): Arcotangente (versión 2 del operando)
- *double* exp(*double* a): Eleva el número 'e' a la potencia 'a'
- *double* log(*double* a): Logaritmo del valor 'a'
- *double* sqrt(*double* a): Raíz cuadrada del valor 'a'
- *double* pow(*double* a, *double* b): Elevar el valor 'a' a la potencia 'b'
- *double* cell(*double* a): Método *ceilling*
- *double* floor(*double* a): Método *floor*
- *double* rint(*double* a): Entero aleatorio
- *int* round(*double* a): Redondea un *double*
- *long* round(*float* a): Redondea un *float*
- *double* random( ): Número aleatorio
- *int* abs(*int* a): Valor absoluto de un entero
- *long* abs(*long* a): Valor absoluto de un *long*
- *float* abs(*float* a): Valor absoluto de un *float*
- *double* abs(*double* a): Valor absoluto de un *double*
- *int* min(*int* a, *int* b): Mínimo de los dos enteros
- *long* min(*long* a, *long* b): Mínimo de dos *long*
- *float* min(*float* a, *float* b): Mínimo de dos *float*
- *double* min(*double* a, *double* b): Mínimo de dos *double*
- *int* max(*int* a, *int* b): Máximo de dos enteros
- *long* max(*long* a, *long* b): Máximo de dos *long*
- *float* max(*float* a, *float* b): Máximo de dos *float*
- *double* max(*double* a, *double* b): Máximo de dos *double*

Comparación de cadenas

Cuando se está trabajando con la clase *String*, hay algunos métodos que se pueden usar como operadores. Por ejemplo, los métodos *equals*, *equalsIgnoreCase* y *compareTo*, como sigue:

- s1.equals(s2): Verdadero si s1 es igual a s2.
- s1.equalsIgnoreCase(s2): Verdadero si s1 es igual a s2 (ignorando las mayúsculas y minúsculas).
- s1.compareTo(s2): Devuelve un valor menor que cero si s1 < s2, cero si s1 es igual a s2 o un valor mayor que cero si s1 > s2.

He aquí un ejemplo en el que se utilizan todos los métodos:

```
public class app
{
    public static void main(String[] args)
    {
        String s1 = "abc";
        String s2 = "abc";
        String s3 = "ABC";
        String s4 = "bcd";
        if (s1.equals(s2)) {
            System.out.println("s1 == s2");
        }
        else {
            System.out.println("s1 != s2");
        }

        if (s1.equalsIgnoreCase(s3)) {
            System.out.println("s1 == s3 al ignorar las mayúsculas");
        }
        else {
            System.out.println("s1 != s3 al ignorar las mayúsculas");
        }

        if (s1.compareTo(s4) < 0){
            System.out.println("s1 < s2");
        }
        else if (s1.compareTo(s4) == 0 ){
            System.out.println("s1 == s2");
        }
        else if (s1.compareTo(s4) > 0 ){
            System.out.println("s1 > s2");
        }
    }
}
```

Este es el resultado del código:

```
C:\>java app
s1 == s2
s1 == s3 al ignorar las mayúsculas
s1 < s2
```

La sentencia *if*

"Hmm", dice el programador novato (PN), "quiero escribir una rutina en Java que calcule el valor absoluto y no sé cómo hacerlo". "Supongo que nunca ha oído hablar del método *Abs* de la clase *Math*", le responde. "¿El qué?" pregunta PN.

Cuando se quiere controlar el flujo del código, es el momento de usar las sentencias condicionales de Java, como la sentencia *if*. Este es el formato general de la sentencia *if*:

```
if (condición) sentencia1:
else sentencia2;
```

Observe que tanto sentencia1 como sentencia2 pueden ser compuestas, es decir, cierto número de sentencias encerradas entre llaves.

Una forma de obtener el valor absoluto, como el programador novato estaba tratando de hacer, es empezar comprobando si el valor es mayor que cero y, si es así, visualizar el resultado tal cual. Aquí, se puede ver cómo se haría esa evaluación con una sentencia *if*:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 10;

        if(value > 0)
            System.out.println("Abs(" + value + ") = " + value);
    }
}
```

Este es el resultado:

```
C:\>java app
Abs(10) = 10
```

Observe que en este caso, la sentencia que se ejecuta si la condición de la sentencia *if* es verdadera es simple, pero también se pueden ejecutar múlti-

ples sentencias haciendo que formen parte de un conjunto de sentencias en un bloque de código, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 10;

        if(value > 0) {
           System.out.println("El número era positivo.");
           System.out.println("Abs(" + value + ") = " + value);
        }
    }
}
```

Este es el resultado:

```
C:\>java app
El número era positivo.
Abs(10) = 10
```

La sentencia *else*

En el ejemplo del valor absoluto, la sentencia *if* sólo visualiza un valor absoluto si el valor ya es positivo.

Sin embargo, se puede extender esa sentencia *if* añadiendo una cláusula *else*, que se ejecuta si la condición de la sentencia *if* es falsa. Así se haría en el código (observe que se pueden gestionar tanto números positivos como negativos):

```
public class app
{
    public static void main(String[] args)
    {
        int value = -10;

        if(value > 0) {
            System.out.println("Abs(" + value + ") = " + value);
        }
        else {
            System.out.println("Abs(" + value + ") = " + -value);
        }
    }
}
```

Este es el resultado del código:

```
C:\>java app
Abs(-10) = 10
```

If anidados

También se pueden tener sentencias *if* anidadas una dentro de otra (esto es, definirlas dentro de otras sentencias *if*). Esto es un ejemplo que muestra cómo funciona la técnica:

```
public class app
{
    public static void main(String[] args)
    {
        double value = 2;

        if (value != 0) {
            if (value > 0)
               System.out.println("El resultado = " + (1 / value));
            else
               System.out.println("El resultado = " + (-1 / value));
        }
    }
}
```

Este es el resultado del código:

```
C:\>java app
El resultado = 0.5
```

Escalas *if-else*

Ès posible crear una secuencia entera de sentencias *if-else*, que se conoce como escala *if-else*. A continuación veremos un ejemplo en el que se ve cómo funciona (en este caso, se evalúa el valor de una variable tipo *string* hasta que se encuentra el día de la semana):

```
public class app
{
    public static void main(String[] args)
    {
        String day = "miércoles";

        if(day == "lunes")
            System.out.println("Es lunes.");
        else if (day == "martes")
            System.out.println("Es martes.");
        else if (day == "miércoles")
            System.out.println("Es miércoles.");
        else if (day == "jueves")
            System.out.println("Es jueves.");
        else if (day == "viernes")
            System.out.println("Es viernes.");
        else if (day == "sábado")
```

```
            System.out.println("Es sábado.");
        else if (day == "domingo")
            System.out.println("Es domingo.");
    }
}
```

Este es el resultado del código:

```
C:\>java app
Es miércoles.
```

Observe que aunque se pueden crear escalas *if-else* de esta forma, Java incluye, para situaciones como esta, la sentencia *switch*. Veremos esta sentencia en el siguiente apartado.

La sentencia *switch*

"Vaya", dice el programador novato (PN), "estoy harto de escribir escalas *if-else*, ya llevo cinco páginas de programa". "¿Y si intenta utilizar la sentencia *switch*?" le pregunta. "¿Qué es eso?" pregunta PN.

La sentencia *switch* es la sentencia de múltiples selecciones de Java; tiene la misma funcionalidad que la escala *if-else* (ver la sección anterior) pero de una forma mucho más fácil. En general, esta es la expresión de la sentencia *switch*:

```
switch (expresión) {
  case valor1:
       sentencia1;
       [break;]
  case valor2:
       sentencia2;
       [break;]
case valor3:
       sentencia3;
       [break;]
            .
            .
            .
  default:
       sentencia_por_defecto;
}
```

En este caso, el valor de la expresión, que debe ser de tipo *byte*, *char*, *short* o *int*, se compara con los distintos valores de las sentencias *case*: valor1, valor2, y así sucesivamente. Si la expresión coincide con una de las sentencias *case*, se ejecuta el código asociado con esa sentencia *case*: senten-

cia1, sentencia2, etc. Si la ejecución llega a una sentencia *break*, la sentencia *switch* termina.

Esto es un ejemplo en el que se visualiza el día de la semana basándose en un valor numérico y utilizando la sentencia *switch*:

```
public class app
{
    public static void main(String[] args)
    {
        int day = 3;

        switch(day) {
            case 0:
                System.out.println("Es domingo.");
                break;
            case 1:
                System.out.println("Es lunes.");
                break;
            case 2:
                System.out.println("Es martes.");
                break;
            case 3:
                System.out.println("Es miércoles.");
                break;
            case 4:
                System.out.println("Es jueves.");
                break;
            case 5:
                System.out.println("Es viernes.");
                break;
            default:
                System.out.println("Debe ser sábado.");
        }
    }
}
```

Este es el resultado:

```
C:\>java app
Es miércoles.
```

También se pueden anidar sentencias *switch*. Observe que si no se especifica la sentencia *break* al final de la sentencia *case*, la ejecución continuará con el código del siguiente *case*. Algunas veces, esto es útil, ya que se quiere ejecutar el mismo código para varios valores:

```
public class app
{
    public static void main(String[] args)
    {
        int temperature = 68;

        switch(temperature) {
```

```
                case 60:
                case 61:
                case 62:
                case 63:
                case 64:
                    System.out.println("Demasiado frío.");
                    break;
                case 65:
                case 66:
                case 67:
                case 68:
                case 69:
                    System.out.println("Frío.");
                    break;
                case 70:
                case 71:
                case 72:
                case 73:
                case 74:
                case 75:
                    System.out.println("Templado.");
                    break;
                default:
                    System.out.println("Probablemente demasiado calor.");
            }
        }

}
```

Este es el resultado del código:

```
C:\>java app
Frío.
```

Bucle *while*

"Bien", dice el programador novato, "tengo problemas otra vez. El gran jefe quiere que haga un programa comercial que calcule factoriales y, ¡ni siquiera sé lo que es un factorial!" "Bien", le dice, "seis factorial, que se escribe como '6!', es igual a 6 x 5 x 4 x 3 x 2 x 1. Y puede escribir su programa con un bucle *while*".

El cuerpo de un bucle *while* (puede ser una sentencia compuesta por un grupo de sentencias simples encerradas entre llaves) se ejecuta mientras una condición lógica sea verdadera. Este es el formato general del bucle *while*:

```
while (condición)
 sentencia
```

Observe que si la condición no es verdadera, el cuerpo del bucle no se ejecutaría ni una vez. A continuación hay un ejemplo con el bucle *while;* en

este caso, se visualiza un valor, se le va restando 1 y después se visualiza el resultado siempre y cuando sea positivo. Cuando el valor llega a 0, el bucle *while* se para porque la condición (value > 0) es falsa:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 10;

        while (value > 0) {
            System.out.println("Valor actual = " + value--);
        }
    }
}
```

Esto es lo que el bucle *while* devuelve:

```
C:\JavaBB\Test>java app
Valor actual = 10
Valor actual = 9
Valor actual = 8
Valor actual = 7
Valor actual = 6
Valor actual = 5
Valor actual = 4
Valor actual = 3
Valor actual = 2
Valor actual = 1
```

A continuación tenemos otro bucle *while* en el que se resuelve el problema del programador novato y se crea un programa que calcula factoriales:

```
public class app
{
    public static void main(String[] args)
    {
        int value = 6, factorial = 1, temp;

        temp = value;   //copia temporal.

        while (temp > 0) {
            factorial *= temp--;
        }

        System.out.println(value + "! = " + factorial);
    }
}
```

Así es como el programa calcula el factorial de 6:

```
C:\>java app
6! = 720
```

A continuación, hay un ejemplo más avanzado. En este caso, se convierte un número a hexadecimal con el operador módulo. Como los dígitos van en orden inverso, se usa el bucle *while* para ponerlos en una pila de Java, que se verá cuando discutamos las colecciones de clases. Después de poner los dígitos en la pila, se les pasa por otro bucle *while* para cambiar el orden y producir el objeto *StringBuffer* que se visualiza:

```
import java.util.*;

public class app
{
    public static void main(String[] args)
    {
        int value = 32, temp = value;
        StringBuffer sb = new StringBuffer();
        Stack st = new Stack();

        while (temp > 0) {
            st.push(String.valueOf(temp % 16));
            temp >>>= 4;
         }

        while(!st.empty()) {
            sb.append(new String((String) st.pop()));
        }

       System.out.println("La conversión de " + value + " es 0x" + sb);
    }
}
```

Esta es la salida del programa:

```
C:\>java app
```

La conversión de 32 es 0x20

Aquí hay algo que puede ser útil: como en Java, las sentencias *null* son válidas, un bucle *while* no tiene por qué tener un cuerpo. A continuación vemos un ejemplo en el que se muestra una forma cruda de calcular la raíz cuadrada de un entero (observe que todo lo que se hace tiene lugar en la parte de la condición del bucle):

```
public class app
{
    public static void main(String[] args)
    {
        int target = 144, sqrt = 1;

        while (++sqrt * sqrt != target) ;

        System.out.println("sqrt(" + target + ") = " + sqrt);
```

```
    }
}
```

Este es el resultado:

```
C:\>java app
sqrt(144) = 12
```

Otro tipo de bucle *while* es el bucle *do-while*, que se tratará en el siguiente apartado.

Bucle *do-while*

El zar de la Programación Exacta dice, "Entonces, en Java hay un bucle *while*. En C++, tenemos un bucle *while* y un bucle *do-while*". "¡Anda!" contesta. "En Java, también tenemos ambos bucles".

El bucle *do-while* es como un bucle *while*, salvo en que la condición es evaluada al final del bucle, no al comienzo. Este es el formato del bucle *do-while* (tenga en cuenta que la sentencia puede estar compuesta por un número de otras sencillas encerradas entre llaves):

```
do
  sentencia
while(condición);
```

La principal razón para usar *do-while* en vez de *while* es que se necesite que el cuerpo del bucle se ejecute al menos una vez. Por ejemplo, este es un caso en el que el valor que se está evaluando no está disponible para el test hasta el final del bucle:

```
public class app
{
    public static void main(String[] args)
    {
        int values[] = {1, 2, 3, 0, 5}, test, index = 0;

        do {
            test = 5 * values[index++];
        } while (test < 15);
    }
}
```

Por otro lado, hay ocasiones en las que se debería usar un bucle *while* en lugar de *do-while*, cuando el cuerpo del bucle no se debería ejecutar ni una vez si la condición no es verdadera. Por ejemplo, echemos un vistazo al siguiente código en el que un bucle *do-while* evalúa el inverso de un valor pero sólo puede probarlo si el valor es distinto de cero al final del bucle:

```
public class app
{
    public static void main(String[] args)
    {
        double value = 0;

        do {
            System.out.println("El recíproco = " + 1 / value);
         } while (value > 0);
    }
}
```

En este caso, es mucho mejor utilizar un bucle *while*:

```
public class app
{
    public static void main(String[] args)
    {
        double value = 0;

        while (value > 0) {
             System.out.println("El recíproco = " + 1 / value);
        }
    }
}
```

Bucle *for*

El programador novato regresa y dice, "Me gustan los bucles *while*, pero cuando se gestionan *arrays* no son los más fáciles de utilizar, realmente necesito un índice numérico. ¿Existe algo así?" "Claro", le dice, "pruebe el bucle *for*".

El bucle *for* de Java es una buena elección cuando se quiere usar un índice numérico que se incremente o decremente automáticamente cada vez que se pase por el bucle, como ocurre cuando se está trabajando con un *array*. En general, este es el formato del bucle *for* (observe que la sentencia puede ser compuesta, incluyendo varias sentencias simples entre llaves):

```
for (expresión_de_inicialización; condición_final; expresión_iterativa)
{
   sentencia
}
```

Se puede inicializar el índice del bucle en la expresión_de_ inicialización (de hecho, se pueden usar múltiples índices en un bucle *for*), proporcionar una condición para terminar el bucle cuando dicha condición sea falsa, en condición_final, y dar alguna forma para cambiar, generalmente incrementando, el índice en expresión_iterativa.

Esto es un ejemplo en el que se pone a funcionar el bucle *for* (observe que el bucle se empieza con el índice a 1 y se termina cuando es mayor que 10, es decir, el cuerpo del bucle se ejecuta exactamente 10 veces):

```
public class app
{
    public static void main(String[] args)
    {
        int loop_index;

        for (loop_index = 1; loop_index <= 10, loop_index++) {
          System.out.println("Esta es la iteración número "
+ loop_index);
        }
    }
}
```

Este es el resultado:

```
C:\>java app
Esta es la iteración número 1
Esta es la iteración número 2
Esta es la iteración número 3
Esta es la iteración número 4
Esta es la iteración número 5
Esta es la iteración número 6
Esta es la iteración número 7
Esta es la iteración número 8
Esta es la iteración número 9
Esta es la iteración número 10
```

Este es un ejemplo que se vio al principio del capítulo; éste calcula el grado medio de un estudiante después de recorrer todos los grados e irlos sumando (observe que se está declarando e inicializando el índice a 0 en la expresión de inicialización):

```
public class app
{
    public static void main(String[] args)
    {
        double grades[] = {88, 99, 73, 56, 87, 64};
        double sum, average;

        sum = 0;

        for (int loop_index = 0; loop_index < grades.length;
            loop_index++) {
            sum += grades[loop_index];
        }

        average = sum / grades.length;

        System.out.println("Grado medio = " + average);
```

```
    }
}
```

Este es el resultado del código:

```
C:\>java app
Grado medio = 77.83333333333333
```

Cuando se declara una variable del bucle (como loop_index en este ejemplo), el alcance de esa variable está limitado al cuerpo del bucle *for* (el alcance de una variable es la parte del programa en la que se puede acceder a ella, como veremos en el siguiente capítulo).

Observe que se pueden usar expresiones muy generales en un bucle *for*. En un bucle *for*, Java permite separar expresiones con una coma, como se muestra en el siguiente ejemplo en el que se utilizan dos índices:

```
public class app
{
    public static void main(String[] args)
    {
        for (int loop_index = 0, doubled = 0, loop_index <=10;
            loop_index++, doubled = 2 * loop_index) {
            System.out.println("El doble del índice " + loop_index +
                " es " + doubled);
        }
    }
}
```

Este es el resultado:

```
C:\>java app
El doble del índice 0 es 0
El doble del índice 1 es 2
El doble del índice 2 es 4
El doble del índice 3 es 6
El doble del índice 4 es 8
El doble del índice 5 es 10
El doble del índice 6 es 12
El doble del índice 7 es 14
El doble del índice 8 es 16
El doble del índice 9 es 18
El doble del índice 10 es 20
```

En un bucle *for* no es obligatorio rellenar todos los elementos; de hecho, se pueden utilizar sentencias *null*. Esto es un ejemplo en el que se suman todos los elementos de un *array* en un bucle *for* que no tiene código en su cuerpo:

```
public class app
{
    public static void main(String[] args)
```

```
    {
        int array[] = {1, 2, 3, 4, 5}, sum = 0;

        for (int loop_index = 0,
            loop_index < array.length;
            sum + = array[loop_index++]) ;

        System.out.println("La suma = " + sum);
    }
}
```

Este es el resultado:

```
C:\>java app
La suma = 15
```

También se puede convertir un bucle *while* en un bucle *for*. A continuación se ha adaptado el ejemplo del factorial del punto anterior "Bucle *while*".

```
public class app
{
    public static void main(String[] args)
    {
        int value = 6, factorial = 1, temp;

        temp = value;    //copia temporal.

        for ( ;temp > 0; ) {
            factorial *= temp--;
        }

        System.out.println(value + "! = " + factorial);
    }
}
```

Bucles anidados

"Estoy trabajando con *arrays* de dos dimensiones", dice el programador novato", y me gustaría tener un bucle dentro de otro para poder recorrer las dos dimensiones". "Por supuesto que puede usar unos bucles dentro de otros", le contesta.

Java permite anidar bucles, uno dentro de otro. Esto es un ejemplo en el que se ve cómo funciona (en este caso, se haya el valor medio de los elementos de un *array* de dos dimensiones recorriendo todos los elementos con dos bucles *for*):

```
public class app
{
```

```
    public static void main(String[] args)
    {
        double array[][] = {{1, 2, 3},
                            {3, 2, 1},
                            {1, 2, 3}};
        int sum = 0, total = 0;

        for(int outer_index = 0; outer_index < array.length;
            outer_index++) {
            for(int inner_index = 0; inner_index <
                array[outer_index].length; inner_index++) {

                sum += array[outer_index][inner_index];
                total++;

            }
        }

        System.out.println("Valor medio del array = " + (sum / total));
    }
}
```

Este es el resultado:

```
C:\>java app
Valor medio del array = 2
```

Sentencia *break*

El programador novato tiene otro problema: "Estoy recorriendo un *array* multidimensional, y algunas veces, al insertarse en cinco bucles anidados, los resultados exceden el máximo valor permitido, por lo que quiero terminar todos los bucles. ¿Cómo lo hago para que todos ellos terminen de forma natural?" "Usando la sentencia *break*", le responde.

Algunos lenguajes incluyen la sentencia *goto* que se puede utilizar para saltarse alguna sentencia del código, pero la mayor parte de los lenguajes consideran que *goto* no es estructurada (Java es de ésos y no incluye la sentencia *goto*). Como Java no tiene una sentencia *goto*, soporta la sentencia *break* con este propósito.

Se puede usar la sentencia *break* para terminar un bucle, como en el siguiente caso, en el que se termina un bucle si la suma es mayor que 12:

```
public class app
{
    public static void main(String[] args)
    {
        double array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int sum = 0;
```

```
        for(int loop_index = 0; loop_index <
            array.length; loop_index++) {

            sum += array[loop_index];
            if (sum > 12) break;
            System.out.println("Recorriendo el bucle...");
        }
        System.out.println("La suma excede el valor máximo.");
    }
}
```

Este es el resultado:

```
C:\>java app
Recorriendo el bucle...
Recorriendo el bucle...
Recorriendo el bucle...
Recorriendo el bucle...
La suma excede el valor máximo.
```

¿Qué ocurre si se quiere salir de varios bucles anidados? En ese caso, se pueden etiquetar los bucles e indicar de cuál se quiere salir. Esto es un ejemplo en el que se sale de un bucle doblemente anidado:

```
public class app
{
    public static void main(String[] args)
    {
        double array[][] = {{1, 2, 3},
                            {3, 2, 1},
                            {1, 2, 3}};
        int sum = 0;

        outer: for(int outer_index = 0; outer_index < array.length;
            outer_index++) {
            inner: for(int inner_index = 0; inner_index <
                array[outer_index].length; inner_index++) {

                sum += array[outer_index][inner_index];
                if (sum > 3) break outer;
            }
            System.out.println("No voy a imprimir.");
        }

        System.out.println("El bucle ha terminado.");
    }
}
```

Este es el resultado:

```
C:\>java app
El bucle ha terminado.
```

Observe que si no se usa una etiqueta con la sentencia *break*, sólo se saldrá del bucle actual.

Sentencia *continue*

"Me gustan los bucles", dice el programador novato, "sólo hay un problema. Algunas veces, cuando estoy recorriendo un bucle, tengo un valor que no quiero usar y querría saltármelo y pasar a la siguiente iteración del bucle sin ejecutar nada. ¿Puedo hacerlo?" "Sí, por supuesto", le responde. "Puede usar la sentencia *continue*".

Para pasar a la siguiente iteración de un bucle, se puede usar la sentencia *continue*. Esto es un ejemplo en el que se obtienen los inversos y se intenta evitar el inverso de 0. Si el índice del bucle actual es 0, se pasa a la siguiente iteración. Este es el código:

```
public class app
{
    public static void main(String[] args)
    {
        for(double loop_index = 5; loop_index > -5; loop_index--) {
            if (loop_index == 0) continue;
            System.out.println("El inverso de " + loop_index +
                " = " + (1 / loop_index));
        }

    }
}
```

Este es el resultado del código (observe que esta salida se ha saltado la línea en la que el código intenta calcular el inverso de 0):

```
C:\>java app
El inverso de 5.0 = 0.2
El inverso de 4.0 = 0.25
El inverso de 3.0 = 0.3333333333333333
El inverso de 2.0 = 0.5
El inverso de 1.0 = 1.0
El inverso de -1.0 = -1.0
El inverso de -2.0 = -0.5
El inverso de -3.0 = -0.3333333333333333
El inverso de -4.0 = -0.25
```

4 Programación orientada a objetos

Este capítulo es común a cualquier programa de Java: programación orientada a objetos (POO). La vimos en el capítulo 1, ya que, sin ella, no se puede escribir código en Java. Ahora que ya conocemos la sintaxis básica de Java, estamos preparados para trabajar, de manera formal, con la programación orientada a objetos.

La programación orientada a objetos es, realmente, otra técnica para implementar el famoso dicho de la programación: "divide y vencerás". La idea es encapsular datos y métodos en objetos, de forma que cada objeto sea semiautónomo, encerrando métodos y datos privados (es decir, internos) y salvándolos del desorden general que les rodea. Así, el objeto puede interactuar con el resto del programa por medio de una interfaz bien definida por sus métodos públicos (es decir, se les puede invocar desde fuera).

La programación orientada a objetos fue creada para gestionar programas más grandes y descomponerlos en unidades funcionales. Esto nos lleva al siguiente paso, que consiste en dividir un programa en subrutinas, ya que los objetos pueden contener múltiples subrutinas y datos. El resultado de encapsular partes de un programa en un objeto es que es concebido como un elemento sencillo y no hay que tratar todo lo que el objeto hace internamente.

Como ya se vió en el capítulo 1, suponga que su cocina está llena de tuberías, bombas, un compresor y todo tipo de interruptores para mantener la

temperatura ideal de la comida. Cada vez que esta temperatura sea demasiado alta, habrá que encender el compresor, abrir las válvulas y empezar a mover las bombas manualmente. Ahora bien, toda esa funcionalidad se puede cubrir con un objeto, un frigorífico, en el que todas esas operaciones se gestionan internamente, realimentándose todas las partes de su interior de forma automática.

Esta es la idea que hay detrás de la encapsulación: parte de un sistema complejo que necesita mucha atención y lo convierte en un objeto que gestiona todo internamente con su propio trabajo y puede ser fácilmente, concebido, como un frigorífico. Si el primer principio de la programación orientada a objetos es "divide y vencerás", el segundo es "fuera de la vista, fuera de la mente".

En Java, la programación orientada a objetos gira sobre algunos conceptos clave: clases, objetos, miembros de datos, métodos y herencia. De forma rápida, estos términos significan:

- Una clase es una plantilla desde la que se pueden crear objetos. La definición de una clase incluye especificaciones formales para la clase y cualquier dato y métodos incluidos en ella.
- Un objeto es una instancia de una clase, al igual que una variable es una instancia de un tipo de dato. Se puede pensar en una clase como el tipo de un objeto y se puede pensar en el objeto como una instancia de una clase. Los objetos encapsulan métodos y variables de instancia.
- Los miembros de datos son esas variables que forman parte de una clase y en ellas se almacenan los datos que usa el objeto. El objeto soporta variables de instancia, cuyos valores son específicos del objeto, y variables de clase, cuyos valores son compartidos entre los objetos de esa clase.
- Un método es una función construida en una clase u objeto. Se pueden tener métodos de instancia y métodos de clase. Con objetos, se usan los métodos de instancia, pero se puede usar un método de clase haciendo referencia, simplemente, a la clase por su nombre, sin requerir ningún objeto.
- Herencia es el proceso que consiste en derivar una clase, llamada clase derivada, de otra, llamada clase base, y se pueden utilizar los métodos de la clase base en la clase derivada.

Todos estos conceptos son importantes para la programación orientada a objetos y entraremos en cada uno de ellos con más detalle.



Truco: si está acostrumbrado a trabajar con la programación orientada a objetos en C++, puede sorprenderle el saber que, aunque los programas Java están orientados a objetos, el soporte orientado a objeto de Java es menor que el disponible en lenguajes como C++. Por ejemplo, los diseñadores de Java decidieron permitir a los programadores sobrecargar métodos pero no operadores (aunque Java sobrecarga por sí solo operadores como + para la clase *String*). Además, Java tampoco soporta los destructores y no soporta directamente la herencia múltiple; en su lugar se utilizan las interfaces de Java.

Clases

En la programación orientada a objetos, las clases proporcionan una especie de plantilla para los objetos. Es decir, si se piensa en una clase como un molde de galletas, los objetos que se crean a partir de ella, son las galletas. Se puede considerar que una clase es un tipo de objeto; se usa una clase para crear un objeto y luego se puede llamar a los métodos del objeto desde este código.

Para crear un objeto, se invoca al constructor de una clase, que es un método que se llama igual que la clase. Este constructor crea un nuevo objeto de la clase. En este libro, ya hemos creado clases; cada vez que se crea un programa Java, se necesita una clase. Por ejemplo, veamos el código necesario para crear una clase llamada *app*, que se almacena en un fichero llamado *app.java* (esta clase crea una aplicación Java):

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Cuando se utiliza el compilador de Java, este fichero, *app.java*, se convierte en el fichero de *bytecode app.class*, que gestiona toda la especificación de la clase *app*.

Entonces, ¿cómo se crean objetos desde las clases? Veamos la siguiente sección.

Objetos

En Java, a un *objeto* se le llama instancia de una clase. Para crear un objeto, se llama al constructor de una clase, que tiene el mismo nombre que

ella. He aquí un ejemplo en el que se crea un objeto de la clase *String*, pasando la cadena que se quiere incluir en ese objeto al constructor de la clase *String*:

```
String s = new String("¡Hola desde Java!");
```

Se verá más sobre la creación de objetos con constructores a lo largo de este capítulo. ¿Qué se hace con un objeto cuando se dispone de otro? Se puede interactuar con él usando sus miembros de datos y métodos; veamos las dos secciones siguientes.

Miembros de datos

Los miembros de datos de un objeto se llaman *miembros de datos de instancia* o *variables de instancia*. Los elementos de datos compartidos por todos los objetos de una clase se llaman *miembros de datos de clase* o *variables de clase*. En este capítulo, se verá cómo se crean variables de instancia y variables de clase. Los miembros de datos pueden hacerse accesibles desde fuera de un objeto, o se puede hacer que sean internos al objeto para usar, de forma privada, los métodos del interior del objeto.

Esto es un ejemplo en el que se muestra cómo se deberían utilizar los miembros de datos de un objeto. Supongamos que se tiene una clase llamada *Data_class*, y se crea un objeto de esta clase llamado *data1*:

```
Data_class data1 = new Data_class("¡Hola desde Java!");
```

Si *Data_class* define un miembro de dato que es accesible públicamente llamado *data*, se puede hacer referencia al miembro de dato de *data1* usando el operador punto (.), como sigue:

```
data1.data
```

Esto significa que se pueden meter los datos en *data1*, de la siguiente forma:

```
Data_class data1 = new Data_class("¡Hola desde Java!");
System.out.println(data1.data);
```

De esta forma, se puede hacer referencia a los miembros de datos de un objeto que lo hace accesible públicamente.

Por otro lado, recordemos que el poder invocar esos datos ocultos es una de las motivaciones de la programación orientada a objetos, y dar acceso a los datos internos de un objeto desde un código externo al mismo no es buena

idea. En su lugar, muchas veces se da acceso a los datos de un objeto a un código externo sólo a través de los métodos del objeto (lo que significa que se puede controlar la interfaz del objeto para el resto del programa, comprobando los valores de los datos antes de que esos valores sean almacenados en los miembros de datos del objeto).

Métodos

Los métodos son funciones de una clase. Generalmente los métodos se dividen en aquellos que se usan internamente en la clase, llamados métodos privados (*private*), los que se usan fuera de la clase, llamados métodos públicos (*public*) y los que son usados por la clase y sus derivadas, llamados métodos protegidos (*protected*).

Los métodos privados son, generalmente, llamados en el interior del objeto por otras partes del mismo. En el ejemplo del frigorífico que propusiimos al principio de este capítulo, el termostato puede invocar un método interno llamado *start_compressor* cuando llegue el momento de enfriar.

Una vez que se tiene un objeto que soporta métodos, se pueden usar los métodos de esos objetos. En el ejemplo siguiente, se usa el método *calculate* para trabajar con los dos valores de *operand1* y *operand2* y almacenar el resultado del cálculo en *result*:

```
Calculator calc1 = new Calculator( );

result = calc1.calculate(operand1, operand2);
```

Java soporta dos tipos de métodos: métodos de clase y métodos de instancia. Los métodos de instancia, como en el ejemplo *calculate*, son invocados en objetos (es decir, los objetos son instancias de una clase). Los métodos de clase, por otro lado, son invocados en una clase. Por ejemplo, la clase *java.lang.Math* tiene un método de clase llamado *sqrt* que calcula una raíz cuadrada, y se puede usar como sigue (no es necesario un objeto):

```
public class app
{
    public static void main(String[] args)
    {
        double value = 4, sqrt;

        sqrt = Math.sqrt(value);

        System.out.println("La raíz cuadrada de " + value + " = "
            + sqrt);
    }
}
```

Esto es lo que se puede ver cuando se ejecuta el código:

```
C:\>java app
La raíz cuadrada de 4.0 = 2.0
```

En este capítulo, aprenderá cómo se crean métodos de clase e instancia.

Antes de entrar en el código, hay un concepto más dentro de la orientación a objetos: la herencia.

Herencia

La herencia es uno de los aspectos de la programación orientada a objetos que se ha definido formalmente. Utilizando la herencia, se puede derivar una nueva clase a partir de una antigua, y la nueva heredará todos los métodos y miembros de datos de la antigua. La clase nueva se llama *clase derivada* y la clase original, *clase base*. La idea es añadir lo que se quiera a la nueva clase para darle más funcionalidad que a la clase base.

Por ejemplo, si se tiene una clase llamada vehículo, se podría derivar una nueva clase llamada *coche* y añadir un nuevo método llamado claxon que visualiza "*beep*" cuando se le llama. De esta forma, se ha creado una nueva clase de la clase base y hemos ampliado esa clase con un método adicional.

La herencia es un tema importante en Java, ya que se puede usar la gran librería de clases disponible, derivando de ellas nuestras clases propias. Veremos cómo utilizar la herencia orientada a objetos en el capítulo siguiente.

Ahora que ya hemos visto los conceptos de POO, es hora de que vayamos a la sección siguiente y revisarlos POO en detalle. Todo este material es esencial para la programación de Java, por lo que conviene profundizar en él hasta que se domine.

Declaración y creación de objetos

El programador novato aparece, preparado para discutir la programación orientada a objetos. "Ya sé todo sobre los objetos", dice, "sólo..." "¿Sólo qué?" le pregunta. "Sólo que no sé cómo crear un objeto en un programa".

Antes de utilizar un objeto, es necesario declararlo. Se pueden declarar objetos de la misma forma que se declaran variables de tipo de datos sencillos, pero se puede usar la clase como tipo de objeto. Además, se puede usar el operador *new* para crear objetos en Java. Veamos un ejemplo que utiliza la clase *String*.



Truco: realmente, no siempre se necesita declarar un objeto antes de utilizarlos. En algunos casos, Java crea un objeto automáticamente, como es el caso de los literales *string*, que Java trata como objetos *String*. Esto quiere decir que expresiones como "¡Hola desde Java!".*length*() son válidas.

Para empezar, declararemos un nuevo objeto, s1, de la clase *String*:

```
public class app
{
    public static void main(String[] args)
    {
        String s1;
        .
        .
        .
}
```

Aunque al declarar una variable simple se crea esa variable, la declaración de un objeto no la crea. Para crear el objeto se puede usar el operador *new* con el siguiente formato en el que se pasan parámetros al constructor de la clase:

```
object = new clase([parámetro1 [, parámetro2...]]);
```

La clase *String* tiene varios constructores, como vimos en el capítulo 2. Se pueden pasar cadenas entre comillas a uno de los constructores de la clase *String*; por lo que se puede crear el nuevo objeto, s1, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        String s1;
        s1 = new String("¡Hola desde Java!");
        .
        .
        .
}
```

Ahora, el nuevo objeto, s1, existe y ya se puede utilizar. Por ejemplo, para convertir todos los caracteres de s1 a minúscula, se puede usar el método *toLowerCase* de la clase *String*:

```
s1.toLowerCase( )
```

Además se pueden combinar la declaración y creación en un solo paso. Esto es un ejemplo en el que se declara un nuevo objeto *String*, s2, creándolo con el operador *new*, todo en una línea:

```
public class app
{
    public static void main(String[] args)
    {
        String s1;
        s1 = new String("¡Hola desde Java!");

        String s2 = new String("¡Hola desde Java!");
        .
        .
        .
}
```

Las clases, con frecuencia, tienen varios constructores, cada uno de los cuales tiene diferentes especificaciones de datos (es decir, diferentes tipos y número de parámetros; el compilador sabe qué constructor es el que se quiere usar por el tipo y por el número de parámetros). En términos de orientación a objetos, estos constructores están *sobrecargados* (se verá la sobrecarga en este capítulo).

Por ejemplo, el constructor de la clase *String* está sobrecargado para coger *arrays* de caracteres, así como cadenas de texto, por lo que se puede crear un objeto nuevo, s3, usando un *array* de caracteres:

```
public class app
{
    public static void main(String[] args)
    {
        String s1;
        s1 = new String("¡Hola desde Java!");

        String s2 = new String("¡Hola desde Java!");

        Char c1[] = {'H', 'o', 'l' 'a' ' ' 'a' 'h' 'í'};
        String s3 = new String(c1);
        .
        .
        .
}
```

Algunas veces las clases tendrán métodos que devuelven objetos, lo que quiere decir que usarán internamente el operador *new* (y nosotros no lo tenemos). Esto es un ejemplo en el que se usa el método *valueOf* de la clase *String*, para convertir un *double* en un objeto *String*:

```
public class app
{
    public static void main(String[] args)
    {
        String s1;
        s1 = new String("¡Hola desde Java!");
```

```
        String s2 = new String("¡Hola desde Java!");

        char c1[] = {'H', 'o', 'l', 'a', ' ', 'a', 'h', 'í'};
        String s3 = new String(c1);

        double double1 = 1.23456789;
        String s4 = String valueOf(double1);
        .
        .
        .
}
```

Además, se puede asignar un objeto a otro, como se puede ver aquí:

```
public class app
{
    public static void main(String[] args)
    {
        String s1;
        s1 = new String("¡Hola desde Java!");

        String s2 = new String("¡Hola desde Java!");

        char c1[] = {'H', 'o', 'l', 'a', ' ', 'a', 'h', 'í'};
        String s3 = new String(c1);

        double double1 = 1.23456789;
        String s4 = String.valueOf(double1);

        String s5;
        s5 = s1;

        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(s4);
        System.out.println(s5);
    }
}
```

Internamente, lo que está ocurriendo realmente es que la referencia al objeto de s1 se copia en s5. En la práctica, esto significa que s1 y s5 se refieren al mismo objeto.

Esto es importante saberlo, porque si se cambian los datos de instancia de s1, también se están cambiando los de s5 y viceversa. Si dos variables hacen referencia al mismo objeto, hay que tener cuidado; muchas referencias a un mismo objeto pueden producir errores que son muy difíciles de detectar. Esto generalmente ocurre porque se cree que se está tratando con diferentes objetos.

Al final del código anterior, se visualizan todas las cadenas creadas. Esto es lo que aparece al ejecutar el programa:

```
C:\>java app
¡Hola desde Java!
¡Hola desde Java!
Hola ahí
1.23456789
¡Hola desde Java!
```

Así es como se declaran y se crean los objetos, de la misma forma que se declaran y crean variables sencillas, con el valor añadido de poder configurar objetos pasando datos a un constructor de la clase. Ya es hora de empezar a crear nuestras propias clases y empezaremos con este proceso en el siguiente punto.

Declarar y definir clases

El programador novato (PN) está nervioso y dice, "¡Lo he conseguido! ¡He creado un objeto, y funciona!" "Bien", le dice, con aprobación, "ahora ¿cómo se crearía una clase?" "Uh-oh", dice el PN "¿cómo se hace?"

En Java, la creación de una clase se realiza en dos pasos: la declaración de la clase y su definición. Con la declaración se dice a Java lo que necesita saber sobre la nueva clase. Esta es la forma general de la declaración de una clase:

```
[access] class nombre de clase [extends ...] [implements ...]
{
    //aquí va la definición de la clase.
}
```

La implementación de la clase es lo que se llama definición de clase, y se hace en el cuerpo de la declaración, como se ve en el ejemplo anterior. Esta es la forma general de una definición y declaración de la clase:

```
 access class nombre de clase [extends ...] [implements ...]
{
    [access] [static] tipo variable_de_instancia1;
         .
         .
         .
    [access] [static] tipo variable_de_instanciaN;

    [access] [static] tipo método1 (lista_de_parámetros)
    {
         .
         .
         .
    }
    .
```

```
    .
    .
    [access] [static] tipo métodoN (lista_de_parámetros)
    {
        .
        .
        .
    }
}
```

Aquí, la palabra clave *static* convierte variables en variables de clases o métodos en métodos de clase (en contraposición a las variables y métodos de instancia), como veremos más tarde. El término *access* especifica la accesibilidad de la clase o de un miembro de clase al resto del programa y puede ser *public*, *private* o *protected*. Además, hay un acceso por defecto si no se especifica ningún tipo; se verá más sobre esto en las páginas siguientes. Se usan las palabras clave *extends* e *implements* con la herencia, como veremos en el siguiente capítulo.

Con un ejemplo, quedará claro. Para empezar, crearemos una clase muy sencilla llamada *printer* que define un método, *print* (ya vimos este ejemplo en el capítulo 1). Cuando se llama al método *print*, se visualiza el mensaje "¡Hola desde Java!" en la pantalla. Así sería la clase:

```
class printer
{
    public void print()
    {
        System.out.println("¡Hola desde Java!);
    }
}
```

Ahora se puede usar el método *print* en otras clases, como en este ejemplo, en el que se está creando un nuevo objeto de la clase *printer* usando el operador *new* y el método *print* de ese objeto en una aplicación llamada *app*:

```
class printer
{
    public void print()
    {
        System.out.println("¡Hola desde Java!");
    }
}

public class app
{
    public static void main(String[ ] args)
    {
        printer printer1 = new printer( );

        printer1.print();
```

```
    }
}
```

Ahora se pone este código en un fichero, *app.java*, se compila y se ejecuta como sigue:

```
C:\>java app
¡Hola desde Java!
```

Tomemos un momento para estudiar este ejemplo; observe que se están declarando y definiendo dos clases, *printer* y *app*, en el mismo fichero. En un fichero sólo una clase puede ser declarada como pública y en este caso, es *app*. Al fichero se le da el nombre después de esa clase, lo que quiere decir que el fichero que la contiene debe ser *app.java*. Sin embargo, se pueden tener tantas clases privadas o protegidas como se quiera dentro del fichero (y Java creará diferentes ficheros con extensión "*.class*" cuando se compile).

También se puede dividir este ejemplo en dos ficheros, uno para cada clase. Este es *printer.java*:

```
class printer
{
    public void print()
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Y este es el nuevo *app.java* (observe que se tiene que importar la clase *printer* para poder utilizarla; para más detalles sobre la importación de clases, ver el capítulo 1):

```
import printer;

public class app
{
    public static void main(String[] args)
    {
        printer printer1 = new printer();

        printer1.print();
    }
}
```

Crear variables de instancia

"Hmm", dice el programador novato (PN), "quiero crear una clase para almacenar datos, y tengo todo menos un pequeño detalle" "¿Sí?" le pregunta. "¿Cómo almaceno datos en la clase?" pregunta el PN.

En la clase, se pueden almacenar datos de dos formas, como variables de instancia o como variables de clase. Las variables de instancia son específicas para los objetos; si se tienen dos objetos (es decir, dos instancias de una clase), las variables de instancia de cada objeto son independientes de las variables de instancia del otro objeto. Por otro lado, las variables de clase de ambos objetos se referirán a los mismo datos y por lo tanto tendrán el mismo valor. En primer lugar, echemos un vistazo a las variables de instancia.

Así es como se almacenan datos de instancia en una clase:

```
access class nombre de clase [extends ...] [implements ...]
{
    [access] tipo variable_de_instancia1;
        .
        .
        .
    [access] tipo variable_de_instanciaN;
}
```

Este es un ejemplo en el que se crea una clase llamada *Data* que gestiona una variable de instancia de tipo *String* llamada *data_string*, que contiene el texto "¡Hola desde Java!":

```
class Data
{
    public String data_string = "¡Hola desde Java!";
}
```

Ahora, se puede crear un objeto, llamado *data*, de la clase *Data* en *main* y hacer referencia a la variable de instancia *data_string* como *data.data_string*. Así es el código:

```
class Data
{
    public String data_string = "¡Hola desde Java!";
}

public class app
{
    public static void main(String[] args)
    {
        Data data = new Data();

        String string = data.data_string;

        System.out.println(string);
    }
}
```

Como se puede observar, se puede acceder a las variables de instancia públicas de un objeto con el operador punto. Sin embargo, recuerde que una

de las motivaciones de la programación orientada a objetos es mantener la privacidad de los datos. Veremos esto con más detalle en la siguiente sección.

Acceso a variables

"Hey", dice el programador novato, "creía que los objetos encapsulaban los datos de forma privada, ¿cómo ese jorobado Johnson ha podido acceder a los datos de mis objetos?" "Porque utilizó un especificador de acceso erróneo para sus datos", le dice.

Se puede usar un especificador de acceso, llamado *access* en el siguiente código, para fijar la visibilidad de los miembros de datos de una clase a lo largo del resto del programa:

```
access class nombre de clase [extends ...] [implements ...]
{
    [access] [static] tipo variable_de_instancia1;
        .
        .
        .
    [access] [static] tipo variable_de_instanciaN;
}
```

Los valores posibles de *access* son *public*, *private* y *protected*. Cuando se declara un miembro de una clase como *public*, es accesible desde cualquier lugar del programa. Si se declara como *private*, sólo es accesible desde la clase de la que es miembro. Si se declara como *protected*, está disponible para la clase actual, otras clases del mismo paquete (se pueden agrupar librerías de clases en paquetes Java; ya se han visto algunos paquetes Java como *java.lang*, y se verá cómo crear paquetes customizados más tarde en el libro), y clases que son derivadas de esa clase. Si no se usa un especificador de acceso, el acceso por defecto es que el miembro de la clase es visible a la clase de la que es miembro, a las clases derivadas de la misma que están en su mismo paquete y a otras clases del mismo paquete. Los detalles se pueden ver en la tabla 4.1.

Por ejemplo, si se quisiera hacer, en el ejemplo de la sección anterior, que la variable de instancia *data_string* fuera privada a la clase *Data*, se podría declarar *private* como sigue:

```
class Data
{
    private String data_string = "¡Hola desde Java!";
}

public class app
```

```
{
    public static void main(String[] args)
    {
        Data data = new Data();

        String string = data.data_string;

        System.out.println(string);
    }
}
```

Ahora, si se intenta acceder a la variable de instancia *data_string* desde otra clase, como se hizo anteriormente en la clase *app*, el compilador de Java devolverá:

```
C:\>javac app.java -deprecation
app.java:12: Variable data_string in class Data not accessible
from class app.
        String string = data.data_string;
                            ^
1 error
```

Tabla 4.1. Alcance del especificador de acceso (x = dentro del alcance).

Ubicación	Private	Sin modificador	Protected	Public
Misma clase	x	x	x	x
Subclase del mismo paquete		x	x	x
No subclase del mismo paquete	x	x	x	
Subclase de otro paquete			x	x
No subclase de otro paquete			x	

Crear variables de clase

"Oye", dice el programador novato, "tengo una nueva clase llamada *counter*, y necesito llevar un contador total en una variable llamada *counter* para todos los objetos de esa clase. ¿Y ahora qué? Estoy hundido". "No se hunda", le dice. "Sólo necesita usar una variable de clase".

El valor de una variable de clase es compartido por todos los objetos de esa clase, lo que significa que será el mismo para todos los objetos. Una variable se declara como estática con la palabra clave *static* (que realmente

especifica la forma en que el valor es almacenado, como dato estático, en contraposición a otras variables, que se almacenan de forma dinámica en pilas):

```
access class nombre de clase [extends ...] [implements ...]
{
    [access] static tipo variable_de_instancia1;
        .
        .
        .
    [access] static tipo variable_de_instanciaN;
}
```

Esto es un ejemplo en el que se crea una clase llamada *data* con una variable de datos de clase llamada *intdata*:

```
class data
{
    public static int intdata = 0;
}
```

Ahora se pueden crear dos objetos de la clase *data*: a y b. Cuando se ponga la variable *intdata* para a con el valor 1, la variable *intdata* para b será también puesta a 1, como se puede ver aquí:

```
class data
{
    public static int intdata = 0;
}

public class app
{
    public static void main(String[] args)
    {
        data a, b;

        a = new data();
        b = new data();

        a.intdata = 1;

        System.out.println("El valor de b.intdata = " + b.intdata);
    }

}
```

Este es el resultado del código:

```
C:\>java app
El valor de  b.intdata = 1
```

Si se necesita ejecutar algún cálculo para inicializar variables estáticas, se puede hacer en un bloque de código estático, que se etiqueta con la palabra

clave *static*; este código se ejecuta sólo una vez, cuando la clase se carga por primera vez:

```
class data
{
    public static int intdata = 1;
    public static int doubledintdata;

    static
    {
        doubledintdata = 2 * intdata;
    }
}


public class app
{
    public static void main(String[] args)
    {
        data a;

        a = new data();

        System.out.println("El valor de a.doubledintdata = " +
            a.doubledintdata);
    }
}
```

Este es el resultado del código:

```
C:\>java app
El valor de a.doubledintdata = 2
```

Crear métodos

"De acuerdo", dice el programador novato, "ya tengo variables de instancia. ¿Hay algo más que aprender sobre clases?" "Bastante", le dice. " Tome una silla y hablemos sobre la creación de métodos".

Llevamos usando métodos desde que visualizamos nuestro primer mensaje con *System.out.println,* por lo que ya está familiarizado con el concepto. Un método es un bloque de código al que se puede transferir el control y por lo tanto, ejecutar ese código. Así es cómo se crean métodos en una clase:

```
access class nombre de clase [extends ...] [implements ...]
{
    [access] [static] tipo método1 (lista de parámetros)
    {
        .
        .
```

```
        .
    }
    .
    .
    .
    [access] [static] tipo métodoN (lista de parámetros)
    {
        .
        .
        .
    }
}
```

Para declarar y definir un método, se puede usar un especificador de acceso (ver el siguiente punto) y especificar el tipo de retorno del método si se quiere que devuelva un valor. Ejemplos de ellos son *int*, *float*, tipo *object* o *void*, si el método no devuelve ningún valor. Se da el nombre del método y se sitúa la lista de parámetros que se le quieren pasar después de ese nombre. El cuerpo actual del método, el código que se ejecutará cuando se le llame, está encerrado en un bloque de código que sigue a la declaración del método.

Veamos un ejemplo. De hecho, ya ha visto uno antes en este capítulo, la clase *printer*. En ese ejemplo, se añadió un método público llamado *print* a la clase *printer*, se creó un objeto de la clase *printer* y se llamó al método *print*, como sigue:

```
class printer
{
    public void print()
    {
        System.out.println("¡Hola desde Java!");
    }
}

public class app
{
    public static void main(String[] args)
    {
        printer printer1 = new printer();

        printer1.print();
    }
}
```

En este caso, el método *print* no tiene parámetros y no devuelve ningún valor, pero se siguen poniendo los paréntesis después del nombre del método; es obligatorio cuando se está llamando a un método en Java ya que es la forma en que el compilador Java sabe que *print* es un método y no un miembro de datos). Esta es la salida del código:

```
C:\>java app
¡Hola desde Java!
```

Hay muchas cosas que conocer sobre la creación de métodos en Java, por lo tanto, vamos a verlas a lo largo de los siguientes puntos. Uno de los aspectos más importantes de los métodos es que se puede hacer que sean puramente internos a un objeto, con el concepto de encapsulación de la programación orientada a objetos, y por ahí es por donde vamos a empezar.

Establecer el acceso a los métodos

"Ese maldito Johnson", dice el programador novato (PN), "ha estado utilizando los métodos internos de mis objetos, aunque yo claramente llamé al método *internal_use_only*. ¿No hay nada más riguroso que pueda utilizar para aislar a ese jorobado?" "Sí", le dice, "puede usar un especificador de acceso más riguroso". "¡Fantástico!" dice PN.

A los métodos de una clase se les puede añadir un especificador de acceso, como sigue (*access* es el especificador de acceso):

```
access class nombre de clase [extends ...] [implements ...]
{
    [access] [static] tipo método1 (lista de parámetros)
    {
        .
        .
        .
    }
    .
    .
    .
    [access] [static] tipo métodoN (lista de parámetros)
    {
        .
        .
        .
    }
}
```

Los valores posibles de *access* son *public*, *private* y *protected*. Cuando se declara un miembro de una clase como *public*, es accesible desde cualquier lugar del programa. Si se declara como *private*, sólo es accesible desde la clase de la que es miembro. Si se declara como *protected*, está disponible para la clase actual, otras clases del mismo paquete y clases que son derivadas de esa clase. Si no se usa un especificador de acceso, el acceso por defecto es que el miembro de la clase es visible a la clase de la que es

miembro, a las clases derivadas de la misma que están en su mismo paquete y a otras clases del mismo paquete. Los detalles se pueden ver en la tabla 4.1.

Esto es un ejemplo en el que se ha añadido un método *private* a la clase *printer* desarrollada en las secciones anteriores. Este método sólo puede ser llamado desde otros métodos de la clase *printer*, como sigue:

```
class printer
{
    public void print()
    {
        internal_use_only();
    }

    private void internal_use_only ()
    {
        System.out.println("¡Hola desde Java!");
    }
}

public class app
{
    public static void main(String[] args)
    {
        printer printer1 = new printer();

        printer1.print();
    }
}
```

Cuando se llama al método *print* de la clase printer, se utiliza el método *internal_use_only*, que no está accesible fuera del objeto, para hacer la visualización. Este es el resultado del código:

```
C:\>java app
¡Hola desde Java!
```

Con frecuencia, es una buena idea hacer que los métodos sean privados o protegidos, ya que se reduce o se controla la accesibilidad del método al resto del código.

Pasar parámetros a los métodos

El especialista en soporte a clientes de la empresa le llama y le dice, "Tenemos un problema". "¿Cuál es el problema?" le pregunta. "Su clase *printer* visualiza un mensaje, pero los clientes se están quejando porque quieren poder fijar el mensaje que se va a visualizar". "No hay problema", le contesta. "Cambiaré el método *print* para que acepte parámetros".

Cuando se declara un método, se puede especificar una lista de parámetros separados por comas, que se pasa al método poniéndola entre paréntesis después del nombre del método:

```
[access] [static] tipo método1 ([tipo nombre_de_parámetro1 [, tipo
nombre_de_parámetro2...]])
{
    .
    .
    .
}
```

Los valores que se pasan al método estarán accesibles en el cuerpo del mismo, usando los nombres que se les ha dado en la lista de parámetros.

Esto es un ejemplo en el que al método *print* se le pasa la cadena a visualizar. El método se declara de forma que Java sepa que aceptará un parámetro, un objeto *String* llamado s:

```
class printer
{
    public void print(String s)
    {
    .
    .
    .
    }
}
```

Ahora, en el cuerpo del método, se puede hacer referencia al objeto *String* que se ha pasado al método *print* como s:

```
class printer
{
    public void print(String s)
    {
        System.out.println(s);
    }
}

public class app
{
    public static void main(String[] args)
    {
        (new printer()).print("¡Hola otra vez desde Java!");
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola otra vez desde Java!
```

Si hay que pasar más de un parámetro, se pueden especificar en la lista, separados por comas:

```
class calculator
{
    int addem(int op1, int op2)
    {
        int result = op1 + op2;
        .
        .
        .
    }
}
```

Se puede llamar a los métodos con literales, variables, *arrays* u objetos, como sigue:

```
calc.addem(1, int1, array1, obj1)
```

Debería observarse que cuando a un método se le pasa una variable sencilla o un literal, el valor de la variable o literal es lo que se le pasa; este proceso se llama *paso por valor*.

Por otro lado, cuando se le pasa un objeto o *array*, realmente se está pasando una referencia a ese objeto o *array* (de hecho, cuando se almacena un *array* u objeto en una variable, lo que realmente se está almacenando es una referencia a los mismos). Por esa razón, el código del método llamado tiene acceso directo al *array* u objeto original, no a una copia, por lo tanto si ese código cambia algún aspecto del *array* u objeto, como puede ser un elemento del *array* o un miembro de datos del objeto, el *array* u objeto originales cambian. Veremos más sobre esto en este capítulo.

Argumentos de la línea de comandos pasados a *main*

En las aplicaciones hay un *array* especial que se pasa como parámetro al método *main*, un *array* de objetos *String* que gestiona los argumentos de la línea de comandos que el usuario especificó cuando inició Java. Por ejemplo, supongamos que se inicia una aplicación de la siguiente forma:

```
C:\>java app Ya es la hora
```

En este caso, el primer elemento del *array* que se pasa a *main* es "Ya", el segundo "es", el tercero "la" y el cuarto "hora".

Esto es un ejemplo en el que se muestra cómo funciona; esta aplicación visualizará todos los argumentos pasados desde la línea de comandos utilizando un bucle sobre el *array String* que se le pasa como parámetro al método *main*:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Argumentos de la línea de comandos...");

        for(int loop_index = 0; loop_index < args.length;
            loop_index++) {

            System.out.println("Argumento " + loop_index +
                " = " + args[loop_index]);
        }
    }
}
```

Así es como funcionaría la aplicación:

```
C:\>java app Ya es la hora
Argumentos de la línea de comandos...
Argumento 0 = Ya
Argumento 1 = es
Argumento 2 = la
Argumento 3 = hora
```

Devolver valores desde los métodos

El programador novato regresa y dice, "bien, Bay otro problema. El gran jefe quiere que cree una clase calculadora que realice operaciones matemáticas. Sé cómo pasar parámetros a los métodos de esa clase, pero..." "¿Sí?" le dice. "No sé devolver los resultados obtenidos después de haber hecho la operación". "Ah", le dice, "use la sentencia *return*".

En un método se utiliza la sentencia *return* para devolver un valor desde un método y en la declaración del método se indica el tipo del valor de retorno.

```
[access] [static] tipo método1 ([tipo nombre_parámetro1 [, tipo
nombre_parámetro2...]])
{
    .
    .
    .
}
```

El tipo del retorno puede ser cualquier tipo de los que Java reconoce, por ejemplo, *int*, *float*, *double*, el nombre de una clase que se ha definido, *int[]* para devolver un *array* de enteros, o *float[]* para devolver un *array* de *float*.

Esto es un ejemplo en el que la clase *calculator* tiene un método llamado *addem* que coge dos parámetros enteros, los suma y devuelve el resultado. Así es como se declara *addem*:

```
class calculator
{
    int addem(int op1, int op2)
    {
    .
    .
    .
    }
}
```

Así se devuelve la suma de los valores pasados a *addem*, usando la sentencia *return*:

```
class calculator
{
    int addem(int op1, int op2)
    {
        return op1 + op2;
    }
}
```

Así funciona la clase *calculator* en un programa:

```
class calculator
{
    int addem(int op1, int op2)
    {
        return op1 + op2;
    }
}

public class app
{
    public static void main(String[] args)
    {
        calculator calc = new calculator();
        System.out.println("addem(2, 2) = " + calc.addem(2, 2));
    }

}
```

Este es el resultado de la aplicación:

```
C:\>java app
addem(2, 2) = 4
```

Crear métodos de clase

"Vaya", dice el programador novato, "he creado mi nueva clase *calculator* con un método estupendo llamado *addem*, pero ¿por qué tengo que meterme en líos creando un objeto de esa clase antes de poder usar el método *addem*? ¿No se puede llamar al método directamente?" "Se puede", le dice, "si se hace que *addem* sea un método de clase en vez de un método de instancia".

Para hacer que un método sea un método de clase, se debe utilizar la palabra clave *static*:

```
class calculator
{
    static int addem(int op1, int op2)
    {
        return op1 + op2;
    }
}
```

Ahora, se puede llamar al método *addem* directamente usando el nombre de la clase, sin crear un objeto. Esto es un ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
        system.out.println("addem(2,2) = " + calculator.addem(2,2));
    }

}
```

Este es el resultado del código:

```
C:\>java app
addem(2, 2) = 4
```

Además se puede usar un método de clase de la forma usual, como un método de un objeto:

```
class calculator
{
    static int addem(int op1, int op2)
    {
        return op1 + op2;
    }
}

public class app
{
    public static void main(String[] args)
    {
```

```
        calculator calc = new calculator();
        system.out.println("addem(2, 2) = " + clac.addem(2, 2));
    }

}
```

Hay que hacer notar que el método *main* en una aplicación se declara estático porque Java debe llamarlo antes de que exista un objeto.

Si se declara un método estático (incluyendo el método *main* de cualquier aplicación), sólo puede llamar a otros métodos estáticos y acceder a datos estáticos. Además, no puede usar las palabras claves *this* y *super*, que hacen referencia al objeto actual y a su padre, respectivamente, como veremos en este y en el siguiente capítulo. En particular, observe que no puede hacer referencia a datos de instancia en un método estático.



Truco: entonces, ¿cómo se llama desde *main* a métodos no estáticos? Esto se hace como se ha venido haciendo a lo largo del libro: se crea un objeto de alguna otra clase en *main* y se llama a los métodos de ese objeto.

Crear métodos de acceso a datos

"Ese maldito Johnson", dice el programador novato (PN), "está fisgoneando otra vez en el código de mis objetos. Pero esta vez, no puedo declarar todo como *private*, porque el resto del código necesita acceder al miembro de datos en cuestión. ¿Qué puedo hacer?" "Puede fijar un método de acceso a datos", le contesta", y restringir el acceso a sus miembros de datos de una forma bien definida". "¡Se lo haré a ese Johnson!" dice PN.

Se puede restringir el acceso a los datos de sus objetos usando métodos de acceso a datos que deben ser invocados para obtener los datos. Esto es un ejemplo en el que se tiene un miembro de datos *String* llamado *data_string*:

```
class data
{
    private String data_string = "¡Hola desde Java!";
    .
    .
    .
}
```

Se puede dar acceso a este miembro de datos privado con dos métodos: *getData* y *setData*. El método *getData* devuelve el valor de la variable privada *data_string*, como sigue:

```
class data
{
    private String data_string = "¡Hola desde Java!";

    public String getData()
    {
        return data_string;
    }
}
```

Sin embargo, el método *setData* restringe el acceso a los datos internos; en particular, escribiremos este método para que el código desde el que se invoca sólo pueda cambiar los datos internos a una nueva cadena si la longitud de la misma es menor que 100. Así sería:

```
class data
{
    private String data_string = "¡Hola desde Java!";

    public String getData()
    {
        return data_string;
    }

    public void setData(String s)
    {
        if (s.length() < 100) {
            data_string = s;
        }
    }
}
```

Ahora se puede utilizar el método *getData* para obtener la cadena interna y el método *setData* para ponerle una nueva. Este es un ejemplo en el que se muestra cómo se usa *getData*:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println((new data()).getData());
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola desde Java!
```

Es buena idea utilizar los métodos de acceso a datos para garantizar el acceso a los datos internos de los objetos. Usando estos métodos, se puede

controlar la interfaz con esos datos y por lo tanto bloquear operaciones que se consideren ilegales.

Crear constructores

"Hmm", dice el programador novato (PN), "sé cómo usar constructores para inicializar los datos de un objeto, como los constructores de la clase *String* que utilizo para meter texto en una cadena, pero..." "¿Sí?" le pregunta. ¿Cómo puedo crear constructores para mis propias clases?" dice PN.

Crear un constructor para una clase es fácil; basta con añadir un método a una clase con el mismo nombre que la clase, sin ningún especificador de acceso ni tipo de retorno. Vamos a ver un ejemplo en el que se añade un constructor que no tiene parámetros a la clase *printer* que hemos desarrollado en este capítulo. A este constructor se le llama cuando se crea un objeto de la clase *printer* y, en este caso, inicializa los datos internos *data_string* a "¡Hola desde Java!" (observe que todavía se necesitan los paréntesis después del nombre del constructor cuando se declara, aunque no lleve parámetros):

```
class data
{
    private String data_string;

    data()
    {
        data_string = "¡Hola desde Java!";
    }

    public String getData()
    {
        return data_string;
    }
}

public class app
{
    public static void main(String[] args)
    {
        System.out.println((new data()).getData());
    }
}
```

Esto es lo que se verá al ejecutar el programa:

```
C:\>java app
¡Hola desde Java!
```

Este constructor es especialmente sencillo porque no utiliza ningún parámetro. En la siguiente sección se explica el constructor con parámetros.

Pasar parámetros a constructores

"De acuerdo", dice el programador novato, "Java está gracioso de nuevo. He hecho un constructor para mi nueva clase, pero realmente, el objeto no está inicializado con los datos que quiero". "Hmm", le contesta; "¿pasó algún dato al constructor?" "Uh-oh", dice PN.

Se pueden pasar datos a los constructores, al igual que se hace con otros métodos. Este es un ejemplo en el que se usa la clase *printer* del punto anterior, pasando la cadena que se va a visualizar al constructor de la clase *printer*:

```
class data
{
    private String data_string;

    data(String s)
    {
         data_string = s;
    }

    public String getData()
    {
        return data_string;
    }
}

public class app
{
    public static void main(String[] args)
    {
        System.out.println((new data("¡Hola desde Java!")).getData());
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola desde Java!
```

El paso de parámetros a un constructor funciona de la misma forma que el paso de parámetros a cualquier método.

Un ejemplo completo de clase

En este apartado se presentará un ejemplo utilizando los conceptos que se han discutido a lo largo del capítulo. El siguiente ejemplo simula la progra-

mación de una pila. Veremos las pilas con más detalle cuando discutamos las colecciones de Java, pero la teoría es sencilla; la programación de una pila funciona como una pila de platos. Cuando se pone un plato en la parte superior de la pila, se está avanzando un elemento en la pila. Cuando se coge un plato de la pila, se está quitando un elemento de la pila. Observe que los platos van en orden inverso, si se ponen los platos 1, 2 y 3, cuando se quiten de la pila, el plato 3 será el que se quite primero, seguido de los platos 2 y 1.

Para usar la clase *stack*, se crea un objeto de la clase, pasando un argumento al constructor que le indica el tamaño de la pila que se quiere (es decir, cuántos enteros se quieren almacenar en ella). El constructor ubica la memoria para la pila en un *array* llamado *stack_data*, y establece un puntero a la pila, *stack_ptr*, que apunta al artículo que actualmente está en la parte superior de la pila (y es realmente el índice que se utilizará con el *array stack_data*).

Luego, se puede utilizar el método *push* de la pila para avanzar un elemento en ésta, que almacena un dato e incrementa el puntero de la pila hasta la siguiente posición del *array*, o se puede usar el método *pop* para quitar un elemento; el método *pop* devuelve el artículo retirado y decrementa el puntero de la pila.

Esta aplicación se llama *stacker.java*; el código añade 10 artículos a la pila y después los retira:

```
class stack {
    private int stack_data[];
    private int stack_ptr;          // stack_ptr = -1 -> la pila está
vacía

    stack(int size)
    {
        stack_data = new int[size];
        stack_ptr = -1;
    }

    public int pop()
    {
        if(stack_ptr == -1)      // Pila vacía - devuelve error
            return 0;
        else                     // Si no, devuelve datos
            return stack_data[stack_ptr--];
    }

    public int push(int push_this)
    {
        if(stack_ptr >= 99)     // La pila está llena - devuelve error
            return 0;
        else {                  // Si no, almacena datos
            stack_data[++stack_ptr] = push_this;
            return 1;
        }
```

```
    }
}

public class stacker {
    public static void main(String args[])
    {
        int popped_value;
        stack stack1 = new stack(100);

        System.out.println("Añadiendo valores ahora...");
        for(int loop_index = 0; loop_index < 10; loop_index++){
            stack1.push(loop_index);
            System.out.println("Valor añadido-> " + loop_index);
        }

        System.out.println("Quitando valores ahora...");
        for(int loop_index = 0; loop_index < 10; loop_index++){
            popped_value = stack1.pop();
            System.out.println("Valor quitado-> " + popped_value);
        }
    }
}
```

Así funciona el programa:

```
C:\>java stacker
Añadiendo valores ahora...
Valor añadido-> 0
Valor añadido-> 1
Valor añadido-> 2
Valor añadido-> 3
Valor añadido-> 4
Valor añadido-> 5
Valor añadido-> 6
Valor añadido-> 7
Valor añadido-> 8
Valor añadido-> 9
Quitando valores ahora...
Valor quitado-> 9
Valor quitado-> 8
Valor quitado-> 7
Valor quitado-> 6
Valor quitado-> 5
Valor quitado-> 4
Valor quitado-> 3
Valor quitado-> 2
Valor quitado-> 1
Valor quitado-> 0
```

Comprender el alcance de las variables

"Hmm", dice el programador novato, "he definido una nueva variable llamada *the_answer* en un método llamado *get_the_answer*, y estaba tratando

de usar esa variable en un método llamado *get_a_clue*, pero Java dice que la variable no está definida". Parece que es cuestión del alcance de la variable, no se pueden usar variables declaradas en un método, en otro método", le contestamos. "¿No se puede?", pregunta PN.

El alcance de una variable consiste en las partes del programa en las que esa variable puede utilizarse, y como se puede ver desde el punto de vista del programador novato, el alcance es un concepto importante.

Java define tres alcances principales: *alcance a nivel de clase*, *a nivel de método* y *a nivel de bloque de código*.

Si se define un miembro de dato en una clase, estará disponible en la clase, y posiblemente más allá, como se ha visto con los especificadores de acceso *private*, *public* y *protected*.

El alcance de un método se inicia cuando el flujo de la ejecución entra en el método y termina cuando dicho flujo lo abandona. Las variables declaradas en el método sólo son visibles en el propio método. Los miembros de datos de la clase también son visibles en los métodos de la clase, como los parámetros pasados a esos métodos.

También se puede definir un alcance local para las variables utilizadas en bloques de código, ya que se pueden declarar variables en esos bloques. Las variables que se declaran en un bloque de código sólo serán visibles en él y en los bloques de código que estén contenidos en el primero.

La forma más fácil de tener esto en mente es saber que las variables no estáticas declaradas en un bloque de código comprendido entre llaves, se crean y almacenan en una pila local cuando se entra en el bloque de código y se destruyen cuando se abandona dicho bloque (por eso se llaman variables dinámicas). Las variables estáticas, por otro lado, se almacenan en la alocación de datos propios del programa, no en cualquier pila, y por tanto no están fuera del alcance. Están cercanas a las variables globales (es decir, variables para todo el programa) que Java permite.

He aquí un ejemplo en el que se muestran varios niveles de alcance (clase, método y bloque de código):

```
class Class
{
    int int1 = 1; //visible para todo el código de la clase.

    public void method(int int2) //visible para todo el código de este
método.
    {
        int int3 = 3;  //visible para todo el código de este método.

        if(int1 != int2) {

            int int4 = 4;  //visible sólo en este bloque de código.
```

```
            System.out.println("int1 = " + int1
                + " int2 = " + int2
                + " int3 = " + int3
                + " int4 = " + int4);
        }
    }
}

public class app
{
    public static void main(String[] args)
    {
        Class c = new Class();

        c.method(2);
    }
}
```

Esto es lo que se ve cuando se ejecuta el código:

```
C:\>java app
int1 = 1 int2 = 2 int3 = 3 int4 = 4
```

Recursividad

El programador novato entra muriéndose de risa y dice, "nunca pensé que el zar de la Programación Exacta me lo dijera: en C++, los métodos se pueden llamar a sí mismos". "También en Java", le contesta. "¿Eh?" dice PN.

Cada vez que se llama a un método en Java, Java sitúa nuevo espacio en su pila interna para todas las variables del método, lo que quiere decir que no hay razón para que no se pueda llamar al mismo método otra vez, pues un nuevo conjunto de variables será alocado en la pila automáticamente. Lo que es más, un método puede llamarse a sí mismo en Java; a esta técnica se la llama *recursividad.*

El ejemplo clásico de recursividad es calcular un factorial, por lo tanto, lo implementaremos aquí. Para calcular el factorial de un entero positivo n, que se escribe como "n!", se calcula lo siguiente:

```
n! = n * (n - 1) * (n - 2) ...* 2 * 1
```

Este proceso se presta a la recursividad fácilmente, porque cada estado de la recursividad puede calcular una operación en la que multiplica el número que se ha pasado por el factorial del número menos 1. Cuando el número llega a 1 después de las sucesivas llamadas, el método vuelve, y el control regresa a través de las sucesivas etapas, ejecutando una multiplicación en

cada etapa hasta que todas las llamadas anidadas han vuelto y se obtiene el factorial.

Así es el código:

```
class calculator
{
    public int factorial(int n)
    {
        if (n == 1) {
            return n;
        }
        else {
            return n * factorial(n - 1);
        }
    }
}

public class app
{
    public static void main(String[] args)
    {
        calculator calc = new calculator();
        System.out.println("6! = " + calc.factorial(6));
    }
}
```

Esto es lo que devolvería el programa:

```
C:\>java app
6! = 720
```

En la práctica, probablemente no se quiera usar la recursividad con mucha frecuencia, pero es bueno saber que está disponible.

Colección *garbage* y gestión de memoria

"Bien", dice el programador novato, "contesta, se aloca más memoria con el operador *new*, pero ¿cómo se libera cuando no es necesaria? ¿Hay un operador *old*?" "¡Qué va!", "Java lo hace todo".

En algunos lenguajes, como C++, se usa el operador *new* para alocar memoria y luego se usa el operador *delete* para liberarla cuando no se la necesita más. Sin embargo, Java no tiene un operador *delete*. Entonces, ¿cómo se libera la memoria alocada cuando ya no es necesaria?

En Java, hay que contar con un proceso ya construido llamado colección *garbage*. Este proceso es automático, aunque no se pueda predecir cuándo va a tener lugar. Java dispondrá de la memoria alocada que no tenga más referencias. Para que funcione la colección *garbage*, se puede poner un artículo a

null (aunque hacer esto tampoco permite predecir cuándo, si llega el caso, la colección *garbage* empezará a funcionar al ejecutar el programa).

Veamos un ejemplo en el que se está creando un nuevo objeto y luego se pone su variable a *null*. Dado que no hay más referencias a ese objeto, el proceso de la colección *garbage* lo liberará más pronto o más tarde. Este es el código:

```
class Data
{
    public int intdata = 0;

    Data()
    {
        intdata = 1;
    }
}

public class app
{
    public static void main(String[] args)
    {
        Data d = new Data();

        //algo de código...

        d = null;

        //más código...
    }
}
```

Recordemos, por tant, que cuando algún elemento de datos, incluyendo objetos y *arrays*, se ha situado con el operador *new*, se pueden poner sus referencias a *null*, y si Java necesita más memoria, comenzará el proceso de la colección *garbage*. Sin embargo, se tiene que tener cuidado y evitar las referencias circulares.



Truco: si ya está familiarizado con C++, puede que se esté preguntando dónde están los punteros en Java, y la respuesta es que no tiene. Los diseñadores de Java omitieron los punteros por razones de seguridad, para asegurarse que los programadores no podían acceder a la memoria más allá de los límites legales. En lugar de punteros, Java usa referencias, que actúan como punteros de forma oculta. Cuando se crea un objeto nuevo, se obtiene una referencia a ese objeto, y cuando se usa esa referencia, Java la quita automáticamente. Así funciona Java: gestiona las referencias (punteros) automáticamente.

Evitar las referencias circulares

La colección *garbage*, liberación de memoria de aquello que no tiene más referencias en el programa, arranca automáticamente. Sin embargo, se deberían evitar las referencias circulares en las que un objeto hace referencia a otro y el segundo al primero.

Cuando se liberan todas las referencias a estos objetos en un programa, cada uno todavía tiene una referencia interna al otro, lo que significa que la colección *garbage* no puede actuar en el otro objeto. Peor todavía es que, como no hay referencias externas al objeto, no se puede llegar a ese objeto para cambiar la situación. Ambos objetos estarán en memoria, consumiendo recursos, hasta que el programa finalice.

Este es un ejemplo en el que se muestra lo que hemos querido decir: en él, la clase a tiene una referencia interna a un objeto de la clase b, y la clase b, una referencia interna a un objeto de la clase a. Cuando el código de *main* pone la referencia, tiene uno de estos objetos a *null*, y estos objetos continuarán ocupando memoria hasta que el programa finalice. Este es el código:

```
class a
{
    b b1;

    a()
    {
        b1 = new b();
    }
}

class b
{
    a a1;

    b()
    {
        a1 = new a();
    }
}

public class app
{
    public static void main(String[] args)
    {
        a obj = new a();

        obj = null;    //¡existen referencias circulares inaccesibles!
    }
}
```

Sólo hay una forma de evitar esto, y es liberar las referencias circulares antes de ir a la aventura. En la práctica, esto generalmente quiere decir que se

ponen las referencias de un objeto a otros objetos a *null* antes de poner la referencia del objeto, en sí mismo, a *null*. Algunas veces, es posible hacer esto en el método *finalize* (para más detalles, ver el siguiente apartado).

Mientras estamos viendo la gestión de memoria, hay que notar que se tiene algún control sobre la alocación de memoria como un todo (ver la opción de la línea de comando –J en el capítulo1, que permite fijar la cantidad de memoria total alocada cuando el programa se ejecuta). En general, Java gestiona la memoria en los programas.

Colección *garbage* y el método *finalize*

"Hmm", dice el programador novato, "entonces Java tiene un recolector de basura que retira de la memoria los artículos que no están referenciados ya. ¿Hay algo más que yo debería saber sobre este proceso?" "Una cosa", le responde. "*Garbage* llama a un método especial, *finalize*, si existe, y se puede usar este método para limpieza de última hora".

Cuando un objeto está dentro del proceso de *garbage* (ver el punto anterior), este recolector llama a un método del objeto llamado *finalize*, si existe. En este método se puede ejecutar el código de limpieza y, con frecuencia, es buena idea liberar cualquier referencia a otros objetos que tenga el objeto actual para eliminar la posibilidad de referencias circulares (también visto en este capítulo).

Esto es un ejemplo:

```
class Data
{
    public int intdata = 0;
    SuperGiantSizeClass sgsc;

    Data()
    {
        intdata = 1;
        sgsc = new SuperGiantSizeClass(100000000);
    }

    protected void finalize()
    {
        sgsc = null;
    }
}

public class app
{
    public static void main(String[] args)
```

```
    {
        Data d = new Data();

        d = null;
    }
}
```

Sobrecarga de métodos

"Todavía estoy trabajando con mi nuevo programa, *SuperDuperMathPro*", dice el programador novato, "y tengo una gran clase llamada *calculator* con un método llamado *addem* que suma dos números. Además, me gustaría sumar tres números juntos, por lo que creo que tendré que escribir un nuevo método". "En absoluto", le dice. "Puede sobrecargar el método *addem* para que maneje dos o tres operandos". "¿Cómo se hace eso?" pregunta PN.

La sobrecarga de métodos es una técnica de la orientación a objetos que permite definir diferentes versiones de un método, todos con el mismo nombre pero con diferentes listas de parámetros. Cuando se usa un método sobrecargado, el compilador de Java sabrá cuál es el que se quiere utilizar por el número y/o tipo de parámetros que se le pasen, y buscará la versión del método con la lista de parámetros correcta.

Veamos un ejemplo. Para sobrecargar un método, sólo hay que definirlo más de una vez, especificando una nueva lista de parámetros en cada una de ellas. Cada lista de parámetros debe ser diferente de cualquier otra de alguna forma, el número de parámetros o el tipo de uno o más de ellos, por ejemplo. Crearemos el ejemplo sobre el que el programador novato estaba preocupado. Primero, se añade una versión del método *addem* a la clase *calculator* que manejará dos operandos:

```
class calculator
{
    int addeem(int op1, int op2)
    {
        return op1 + op2;
    }
    .
    .
    .
}
```

Luego se añade otra versión del mismo método que acepte tres operandos:

```
class calculator
{
    int addem(int op1, int op2)
    {
```

```
        return op1 + op2;
    }

    int addem(int op1, int op2, int op3)
    {
        return op1 + op2 + op3;
    }
}
```

Ahora se pueden usar ambos métodos en el código, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        calculator calc = new calculator();

        System.out.println("addem(2, 2) = " + calc.addem(2, 2));
        System.out.println("addem(2, 2, 2) = " + calc.addem(2, 2, 2));
    }

}
```

Este es el resultado del programa:

```
C:\>java app
addem(2, 2) = 4
addem(2, 2, 2) = 6
```

Como se puede ver, la sobrecarga proporciona una técnica potente, especialmente en el código que se entrega a otros desarrolladores, porque permite pasar diferentes listas de parámetros a un método, haciéndolo más fácil de usar de diferentes formas en el código.

Además se pueden sobrecargar los constructores; (ver el siguiente punto para más detalles).

Sobrecarga de constructores

Dice el programador novato: "Entonces, ¡se pueden sobrecargar métodos en Java para que manejen diferentes listas de parámetros! ¿También se pueden sobrecargar constructores?" "Por supuesto", le dice. "Considere la clase *String* de Java, que tiene un constructor al que se le pueden pasar cadenas, *arrays* de caracteres y otro tipo de datos". " Es cierto", dice PN.

La sobrecarga de constructores funciona como la sobrecarga de otros métodos (ver el punto anterior para más detalles): sólo hay que definir el

constructor un número de veces, cada una con una lista de parámetros diferentes.

Veamos un ejemplo que imita los constructores de la clase *String* de Java en el que esta nueva clase, la clase *data*, tendrá un constructor al que se le puede pasar un *array* de caracteres o una cadena. Esta clase simplemente almacenará los datos que se le pasen y hará que los datos estén disponibles con un método *getData*.

Así es cómo se declara y define el constructor que toma un *array* de caracteres:

```
class data
{
    private String data_string;

    data(char[ ] c)
    {
        data_string = new String(c);
    }
}
```

Así es como se declara el constructor que acepta una cadena de texto:

```
class data
{
    private String data_string;

    data(char[] c)
    {
        data_string = new String(c);
    }

    data(String s)
    {
        data_string = s;
    }
}
```

Lo único que queda es añadir el método *getData*:

```
class data
{
    private String data_string;

    data(char[] c)
    {
        data_string = new String(c);
    }

    data(String s)
    {
        data_string = s;
```

```
    }

    public String getData()
    {
        return data_string;
    }
}
```

Ahora se pueden usar ambos constructores en el código, crear objetos y visualizar el texto almacenado, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        char chararray[] = {'H', 'o', 'l', 'a'};
        System.out.println((new data(chararray)).getData());

        System.out.println((new data("¡Hola desde Java!")).getData());
    }
}
```

Este es el resultado del código:

```
C:\>java app
Hola
¡Hola desde Java!
```

Pasar objetos a métodos

El programador novato aparece y dice, "Otra vez Java está en plan gracioso. Pasé un objeto a un método porque quería hacer una prueba de destrucción en ese método, pero al devolver el control, el objeto original estaba destruido. ¿Qué ha ocurrido?" "Java pasa objetos por referencia, " le contesta. "Eso es lo que ha ocurrido".

Cuando se pasa un elemento de un tipo de dato sencillo a un método, Java pasa una copia de los datos, que se llama *paso por valor*. Dado que el método sólo obtiene una copia del elemento de los datos, el código del método no puede afectar al elemento del dato original.

Este es un ejemplo en el que se pasa un objeto de la clase *printer* para visualizar los datos del objeto:

```
class Data
{
    public String data_string;

    Data(String data)
    {
```

```
        data_string = data;
    }
}

class printer
{
    public void print (Data d)
    {
        system.out.println(d.data_string);
     }
}

public class app
{
    public static void main(String[] args)
    {
        Data data = new Data("¡Hola desde Java!");
        printer p = new printer();

        p.print(data);
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola desde Java!
```

Como se mencionó anteriormente, dado que los objetos se pasan por referencia, el objeto pasado cambia el objeto original. Aquí tenemos un ejemplo en el que se pasa el objeto de la clase *Data* a un método llamado *rewrite* que cambia la variable de instancia *data_string* del objeto; esta variable empieza con la cadena "¡Hola desde Java!", pero el método *rewrite* es capaz de cambiar la cadena a "¡Hola a Java!" en este código:

```
class Data
{
    public String data_string;

    Data(String s)
    {
        data_string = new String(s);
    }
}

class Class
{
    public void rewrite(Data d)
    {
        d.data_string = "¡Hola a Java!");
     }
}

public class app
```

```
{
    public static void main(String[] args)
    {
        Data d = new Data("¡Hola desde Java!");
        Class c = new Class();

        c.rewrite(d);

        System.out.println(d.data_string);
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola a Java!
```

Pasar *arrays* a métodos

"Entonces, en Java, las variables simples se pasan por valor y los objetos por referencia; ahora ya está todo encaminado", dice el programador novato. "Todavía no", contestamos. "Hay un tipo más de elemento que se pasa por referencia, los *arrays*".

Se puede pasar un *array* a un método tan fácil como se pasa una variable simple, pero hay que tener en cuenta que los *arrays* se pasan por referencia, no por valor, lo que significa que si se cambia un *array* pasado a un método, el *array* original también se ve afectado.

Veamos un ejemplo. En este caso, se pasará un *array* a un método llamado *doubler* que duplica cada elemento del *array*. Dado que los *arrays* se pasan por referencia, los datos del *array* original también serán duplicados. Este sería el código (observe que se visualiza un *array* antes y después de llamar al método *doubler*):

```
class Calculate
{
    public void doubler(int a[ ])
     {
        for (int loop_index = 0; loop_index < a.length;
            loop_index<++) {
            a[loop_index] *=2;
        }
      }
}

public class app
{
    public static void main(String[] args)
```

```
    {
        int array[] = {1, 2, 3, 4, 5};
        Calculate c = new Calculate();

        System.out.println("Antes de llamar a doubler...");
        for (int loop_index = 0; loop_index < array.length;
            loop_index++) {

            System.out.println("array[" + loop_index + "] = " +
                array[loop_index]);
        }

        c.doubler(array);

        System.out.println("Después de llamar a doubler...");
        for (int loop_index = 0; loop_index < array.length;
            loop_index++) {

            System.out.println("array[" + loop_index + "] = " +
                array[loop_index]);
        }
    }
}
```

Este es el resultado del código:

```
C:\>java app
Antes de llamar a doubler...
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
Después de llamar a doubler...
array[0] = 2
array[1] = 4
array[2] = 6
array[3] = 8
array[4] = 10
```

Usar la palabra clave *this*

Los objetos de Java incluyen un miembro de datos llamado *this*, que realmente es una referencia al objeto actual. La palabra clave *this* es útil si se necesita hacer referencia al objeto actual; por ejemplo, cuando se quiere pasar el objeto actual a un método.

En el siguiente caso, la clase *Data* tiene un método, *printData*, que imprime los datos del objeto actual pasando el objeto al método *print* de otro objeto. La palabra clave *this* se usa para hacer referencia al objeto actual. Así sería el código:

```
class Data
{
    private String data_string;

    Data(String s)
    {
        data_string = s;
    }

    public String getData()
    {
        return data_string;
    }

    public void printData()
    {
        printer p = new printer();
        p.print(this);
    }
}

class printer
{
   void print(Data d)
   {
         System.out.println(d.getData());
   }
}

public class app
{
    public static void main(String[] args)
    {
        (new Data("¡Hola desde Java!")).printData();
    }
}
```

Observe que cuando se llama a *p.print*, se pasa una referencia al objeto actual a *p.print*, de forma que el código de *p.print* da acceso al método *getData* del objeto actual, que devuelve los datos internos que se van a visualizar. Este es el resultado del programa:

```
C:\>java app
¡Hola desde Java!
```

Devolver objetos desde métodos

Se pueden devolver objetos desde métodos, igual que ocurre con otros tipos de datos. Sin embargo, surge una pregunta: cuando el método que

devolvió el objeto se queda fuera del alcance, ¿el objeto que él devolvió también se queda fuera del alcance?

La respuesta es no. Cuando se crea un nuevo objeto usando el operador *new*, ese objeto no es destruido cuando el método que lo creó queda fuera del alcance, y el objeto, por sí mismo, no está a disposición del recolector de basura hasta que no haya más referencias a él.

En el ejemplo siguiente, la clase llamada *ObjectFactory* tiene un método llamado *getNewObject* que devuelve un objeto de la clase *CreatedClass*:

```
class ObjectFactory
{
    public CreatedClass getNewObject()
    {
        return new CreatedClass();
    }
}

class CreatedClass
{
    public String tag = "Esta es la etiqueta de datos.";
}

public class app
{
    public static void main(String[] args)
    {
        ObjectFactory o = new ObjectFactory();
        CreatedClass c = o.getNewObject();

        System.out.println(c.tag);
    }
}
```

Cuando se llama al método *getNewObject*, éste devuelve un nuevo objeto de la clase *CreatedClass* y puede visualizar los datos de ese objeto. Esto es lo que muestra el programa al ejecutarlo:

```
C:\>java app
Esta es la etiqueta de datos.
```

Devolver *arrays* desde métodos

Se pueden devolver *arrays* desde métodos igual que se pueden devolver objetos y tipos de datos sencillos. Veamos un ejemplo en el que se crea una clase llamada *ArrayFActory* con un método llamado *getNewArray*. Cuando se llama a *getNewArray*, éste devolverá un *array* de enteros. Observe que el

tipo de retorno que se especifica en la declaración de *getNewArray* es int[], lo que indica que es un *array* de enteros. Este es el código:

```
class ArrayFactory
{
    public int[] getNewArray()
    {
        int array[] = {1, 2, 3, 4, 5};

        return array;
    }
}
```

Así funcionaría la clase *ArrayFactory*, creando un nuevo *array* y visualizándolo:

```
public class app
{
    public static void main(String[] args)
    {
        ArrayFactory af = new ArrayFactory();

        int array[] = af.getNewArray();

        for (int loop_index = 0; loop_index < array.length;
            loop_index++) {

            System.out.println("array[" + loop_index + "] = " +
                array[loop_index]);
        }
    }
}
```

Este es el resultado del código:

```
C:\>java app
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
```

5 Herencia, clases internas e interfaces

Este capítulo trata la herencia, tema muy importante en la programación con lenguaje Java. Usando la herencia se puede derivar una clase, llamada *clase derivada o subclase*, de otra, llamada *clase base o superclase*. Se trata de añadir lo que se quiera a la nueva clase para darle mayor funcionalidad que a la clase original.

El capítulo anterior empezó con la discusión sobre la programación orientada a objetos y, como se mencionó allí, si se tiene una clase llamada, supongamos, "vehículo", que contiene la funcionalidad básica de algunos elementos de transporte, se puede usar esa clase como base de todas aquellas que se deriven de la misma, como "coche" y "camión". La clase "coche" puede, por ejemplo, tener un miembro de datos llamado "ruedas", inicializado a 4, pero el mismo miembro de datos en la clase "camión" debería estar inicializado a 18. Además se puede usar la misma clase "vehículo" como la clase base para otras, como puede ser la clase "helicóptero". Todas las subclases tendrán acceso a los miembros no privados de la superclase y podrán añadir los suyos propios. De hecho, pueden sobrescribir los miembros no privados de la superclase, sustituyéndolos con su propio código. Por ejemplo, la clase "vehículo" podría tener un método llamado *go* que visualice "Conducir...", y la clase helicóptero puede sobrescribir ese método, redefiniéndolo y visualizando "Volar...".

Entonces, por medio de la herencia, puede basar sus clases en otras clases, reutilizando y añadiendo código. Se puede usar o redefinir los miembros de la superclase como guste, personalizado esa clase para su propio uso. De hecho, se pueden crear clases que deban ser tratadas como superclases. Estas clases se llaman *abstractas*. En un objeto, no se puede instanciar directamente una clase abstracta; en su lugar se debe derivar una nueva clase de la primera, sobrescribiendo los miembros que son específicamente declarados como abstractos. Las clases abstractas se utilizan para forzar a los desarrolladores a customizar algunos o todos los miembros de una clase; por ejemplo, se puede tener un método abstracto llamado *printError*, porque se quiere que los desarrolladores suministren su propio código para este método, de forma que sea apropiado para las subclases que ellos crean.

Esta es una visión rápida de lo que hace la herencia. La siguiente pregunta es, ¿por qué es tan importante en Java?

¿Por qué la herencia?

Java es verdaderamente un lenguaje orientado a objetos, y está muy relacionado con la herencia. Los desarrolladores de Sun Microsystems han creado enormes paquetes, librerías de clases, llenos de clases que se pueden usar como superclases. Esto es importante si, por ejemplo, se quiere crear una *applet* en Java, porque en ese caso, se puede derivar la *applet* de la clase *Applet* del paquete *java.applet*. Aquí está la *applet* que se vio en el capítulo 1, que crea una superclase basada en la clase *Applet* usando la palabra clave *extends* (en el siguiente capítulo se verá más sobre *applets*):

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}
```

Este es otro ejemplo que se vio en el capítulo 1; en este caso, se crea una aplicación ventana basándose en la clase de Java *java.awt.Frame*:

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

class AppFrame extends Frame
```

```
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}

public class app
{
    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new WindowAdapter() { public void
           windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
    }
}
```

Como se puede ver, cuando se manejan elementos visuales en los programas, hay que tener mucho contacto con los paquetes de Java. Los botones, por ejemplo, tienen sus propias clases y para personalizados, se derivan sus clases. De hecho, si se quiere gestionar las acciones del ratón o los clics en los botones, se tiene que usar la herencia, y esta vez, no se usan las superclases, sino las interfaces.

¿Por qué las interfaces?

Supongamos que se quiere crear una *applet* que gestione los clics que se hacen en un botón.

Para crear una *applet* estándar, se puede derivar una clase de la clase *java.applet.Applet* y para gestionar los clics que se hacen en un botón, se usa otra clase, *ActionListener*. Por lo tanto, la *applet* se tendrá que basar en ambas clases *Applet* y *ActionListener*.

Basar una subclase en dos o más superclases se llama *herencia múltiple*, y Java no soporta la herencia múltiple (aunque sí lo hacen lenguajes como C++). En la práctica, esto quiere decir que sólo se puede usar la palabra clave *extends* con una clase. Para resolver este problema, Java implementa las clases como *ActionListener* como interfaces.

En el siguiente caso, eso significa que la *applet* se puede extender de la clase *Applet* y usar la palabra clave *implements* para añadir la gestión del clic en los botones. Así quedaría la *applet*:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class clicker extends Applet implements ActionListener {

     TextField text1;
     Button button1;

     public void init(){
               text1 = new TextField(20);
               add(text1);
               button1 = new Button("¡Hacer clic aquí!");
               add(button1);
          button1.addActionListener(this);
        }

        public void actiopnerformed(ActionEvent event){
               String msg = new String ("Bienvenido a Java");
               if(event.getSource() == button1){
                        text1.setText(msg);
                }
        }
}
```

Se puede implementar tantas interfaces como se quiera; por ejemplo, aquí hay parte de un programa que implementa tres *listeners* para permitir que el programa gestione los clics en los botones y las acciones del ratón:

```
import java.awt.Graphics;
import java.awt.*;
import java.awt.event.*;
import java.lang.Math;
import java.applet.Applet;

public class dauber extends Applet implements ActionListener,
MouseListener, MouseMotionListener {
     .
     .
     .
```

Más adelante, se verá cómo se crean las interfaces. Sin embargo, veremos una introducción a las mismas en este capítulo para que podamos utilizarlas después.

Todavía queda un punto por cubrir en este capítulo: las clases internas.

¿Por qué las clases internas?

Java permite crear clases dentro de clases, y la clase encerrada se llama clase *interna*. Empezaremos a trabajar con ellas en este capítulo. Quizás no

vea mucha necesidad de definir clases dentro de clases en este momento, pero lo verá más claro cuando se empiecen a gestionar los eventos de la interfaz de usuario, como, por ejemplo, cuando el usuario cierra una ventana.

Los eventos se gestionan con las interfaces, y cuando se implementa una interfaz, también se debe proporcionar la implementación de varios métodos abstractos en el interior de la misma. Para que este proceso sea más fácil, Java proporciona las clases *adaptador*, que ya tienen implementaciones vacías de los métodos requeridos. Para gestionar un evento de interfaz de usuario de cualquier tipo, es común tener una subclase de las clases adaptador como clase interna, sobrescribiendo los métodos que se quiera de una forma muy compacta. Este es un ejemplo en el que el código finaliza una aplicación cuando el usuario cierra la ventana de la misma:

```
public class app
{
    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new windowAdapter() {public void
           windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
    }
}
```

Seguiremos este código con detalle cuando trabajemos con eventos, y más tarde, introduciremos clases internas para que este código tenga mucho más sentido.

Ahora que ya conocemos los conceptos relacionados con la herencia, las interfaces y las clases internas, es hora de empezar con la siguiente sección.

Crear una subclase

"De acuerdo", dice el programador novato, "quiero aprender todo lo que hay sobre la herencia. ¿Puede explicármelo en dos palabras o menos?" Mientras cuenta con los dedos, le dice, "No hay forma".

Veamos un ejemplo en el que se muestra cómo se crea una subclase usando le herencia. Suponga que tiene una clase llamada "vehículo" que tiene un método *start,* que se puede utilizar para arrancar el vehículo y que visualiza "Arrancar...":

```
class vehiculo
{
    public void start()
    {
        System.out.println("Arrancar...");
    }
}
```

Hay otro tipo de vehículos, por lo tanto si se quiere especializar la clase "vehículo" en una clase "coche", se puede usar la herencia con la palabra clave *extends*. Así se declara "coche" como una subclase de "vehículo":

```
class coche extends vehiculo
{
    .
    .
    .
}
```

La sintaxis indica que "coche" es derivada de la clase "vehículo", lo que significa que en este caso, "coche" heredará el método *start* de "vehículo". Además se pueden añadir los miembros de datos y métodos propios en las subclases. Este es un ejemplo en el que se añade un método llamado *drive* a la clase "coche":

```
class coche extends vehiculo
{
    public void drive()
    {
        system.out.println("Conducir...");
    }
}
```

Ahora se puede acceder a ambos métodos *start* y *drive* en los objetos de la clase "coche", como se ve en el ejemplo:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un coche...");

        Coche c = new coche();
        c.start();
        c.drive();
    }
}
```

Esta es la salida del código anterior:

```
C:\>java app
Crear un coche...
```

```
Arrancar...
Conducir...
```

Este es un ejemplo básico de subclase. Hay mucho más en todo este proceso. Por ejemplo, ¿qué pasa si se define un método en una subclase con el mismo nombre que un método de la superclase? ¿Cómo se pasan los datos al constructor de una superclase? Todo esto lo veremos en el siguiente capítulo.

Especificadores de acceso y herencia

"Otra vez, Java está gracioso", dice el programador novato. "En una subclase, quiero usar algunos métodos de la superclase, pero Java dice que no existen" "¿Cuál es el especificador de acceso para los métodos?" le pregunta. "Privado", dice PN. "Ese es el problema".

Los especificadores de acceso se usan con clases, miembros de datos y métodos para especificar la visibilidad de esos elementos en el resto del programa. Esta es la forma general de declarar y definir las clases, que muestra cómo se usan los especificadores de acceso:

```
access class nombre de clase [extends ...] [implements ...]
{
    [access] [static] tipo variable_de_instancia1;
         .
         .
         .
    [access] [static] tipo variable_de_instanciaN;

    [access] [static] tipo método1 (lista de parámetros)
    {
         .
         .
         .
    }
    .
    .
    .
    [access] [static] tipo métodoN (lista de parámetros)
    {
         .
         .
         .
    }
}
```

Los valores posibles de *access* son *public*, *private* y *protected*. Cuando se declara un miembro de una clase como *public*, es accesible desde cualquier

lugar del programa. Si se declara como *private*, sólo es accesible desde la clase de la que es miembro. Si se declara como *protected*, está disponible para la clase actual, otras clases del mismo paquete, y clases que son derivadas de esa clase. Si no se usa un especificador de acceso, el acceso por defecto es que el miembro de la clase es visible a la clase de la que es miembro, a las clases derivadas de la misma que están en su mismo paquete y a otras clases del mismo paquete. Se pueden ver los detalles en la tabla 5.1.

Por ejemplo, echemos un vistazo al código del programador novato, en el que el método *start* se declara como *private* pero además se accede a él en *main*:

```
class vehiculo
{
    private void start()
    {
        System.out.println("Arrancar...");
    }
}

class coche extends vehiculo
{
    public void drive()
    {
        System.out.println("Conducir...");
    }
}

public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un coche...");
        coche c = new coche();
        c.start();
        c.drive();
    }
}
```

Tabla 5.1. Alcance para el especificador de acceso (x = en el alcance).

Ubicación	*Private*	*Sin modificador*	*Protected*	*Public*
Misma clase	x	x	x	x
Subclase en el mismo paquete		x	x	x
No subclase en el mismo paquete		x	x	x

Ubicación	Private	Sin modificador	Protected	Public
Subclase en otro paquete			x	x
No subclase en otro paquete				x

Dado que la declaración de un método, usando *private,* restringe ese miembro a su clase, Java dice que no puede encontrar el método *start* cuando se usa en *main*:

```
C:\>c:\jdk1.2.2\bin\javac app.java -deprecation
app.java:47: No method matching start() found in class coche.
        c.start();
          ^
1 error
```

Por otro lado, la declaración de un miembro como *protected* restringe su alcance al código del mismo paquete y de la subclase de la clase en la que está declarado. Sin embargo, el siguiente código funciona:

```
class vehiculo
{
    protected void start()
    {
        System.out.println("Arrancar...");
    }
}

class coche extends vehiculo
{
    public void drive()
    {
        System.out.println("Conducir...");
    }
}

public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un coche...");
        coche c = new coche();
        c.start();
        c.drive();
    }
}
```

Esta es la salida de la aplicación:

```
C:\>java app
Crear un coche...
```

```
Arrancar...
Conducir...
```

Llamar a los constructores de la superclase

"De acuerdo", dice el programador novato, "tengo un problema. Sé que puedo crear una subclase a partir de una superclase, pero ¿qué pasa si la superclase tiene un constructor?" "Hay que pensar un poco". "Hmm", dice el PN, pensativo.

Supongamos que tiene una clase llamada a que tiene un constructor sin parámetros:

```
class a
{
    a()
    {
        System.out.println("En el constructor de a...");
    }
}
```

Luego se deriva una subclase b de a:

```
class b extends a
{
}
```

Ahora, cuando se crea un objeto de la clase b, se llama automáticamente al constructor de la clase a:

```
public class app
{
    public static void main(String[] args)
    {
        b obj = new b();
    }
}
```

Este es el resultado del código precedente:

```
C:\>java app
En el constructor de a...
```

Ahora, supongamos que se añade un constructor sin parámetros a la clase b:

```
class a
{
    a()
    {
```

```
        System.out.println("En el constructor de a...");
    }
}

class b extends a
{
    b()
    {
        System.out.println("En el constructor de b…");
    }
}

public class app
{
    public static void main(String[] args)
    {
        b obj = new b();
    }
}
```

En este caso, cuando se crea un objeto de la clase b, se llama a los constructores de a y b:

```
C:\>java app
En el constructor de a…
En el constructor de b…
```

Ahora supongamos que se cambia el constructor de b para que acepte un parámetro:

```
class a
{
    a()
    {
        System.out.println("En el constructor de a...");
    }
}

class b extends a
{
    b(String s)
    {
       System.out.println("En el constructor String de b…");
       System.out.println(s);
    }
}

public class app
{
    public static void main(String[] args)
    {
        b obj = new b("¡Hola desde Java!");
    }
}
```

En este caso, se llama a los constructores de ambas clases, a y b:

```
C:\>java app
En el constructor de a...
En el constructor String de b...
¡Hola desde Java!
```

Un constructor sin parámetros se llama *constructor por defecto* de una clase, ya que Java lo llamará automáticamente cuando se instancien subclases de esa clase, a menos que se haga otro tipo de peticiones. ¿Qué significa eso? Supongamos que se añade otro constructor a la clase a para gestionar las cadenas (esto se llama sobrecarga, y lo veremos más tarde en este capítulo):

```
class a
{
    a()
    {
        System.out.println("En el constructor de a...");
    }

    a(String s)
    {
        System.out.println("En el constructor String de a…");
        System.out.println(s);
    }
}
```

Ahora digamos que se quiere llamar al constructor de la clase a con el parámetro *String* en lugar de llamar a los constructores por defecto. ¿Cómo se hace esto? Se llama al método *super* del constructor de la clase b, como sigue:

```
class a
{
    a()
    {
        System.out.println("En el constructor de a...");
    }

    a(String s)
    {
        System.out.println("En el constructor String de a...");
        System.out.println(s);
    }
}

class b extends a
{
    b(String s)
    {
        super(s);
        System.out.println("En el constructor de b...");
```

```
        System.out.println(s);
    }
}

public class app
{
    public static void main(String[] args)
    {
        b obj = new b("¡Hola desde Java!");
    }
}
```

Ahora cuando se instancia un objeto de la clase b, se llama al constructor de la clase a que tiene un parámetro *String*, no al constructor por defecto:

```
C:\>java app
En el constructor String de a...
¡Hola desde Java!
En el constructor String de b...
¡Hola desde Java!
```

¿Por qué se llama al constructor que tiene un parámetro *String* en la clase a y no al constructor por defecto de la clase a? La razón es sencilla, Java ve que se está usando el método *super* para llamar al método de la superclase:

```
class b extends a
{
    b(String s)
    {
        super(s);
        System.out.println("En el constructor String de b...");
        System.out.println(s);
    }
}
```

Si se usa *super* para llamar al constructor de una superclase, la línea en la que se hace esto debe ser la primera del constructor, que es donde Java lo buscará (si es cualquier otra línea distinta de la primera, Java lo ignora).

Crear multiniveles de herencia

"Entonces", dice el programador novato, "¿se pueden crear subclases de subclases?" "Sí", le dice. "Y ¿se puede crear subclases de subclases de subclases?" " Sí", le dice. PN está dispuesto a continuar y pregunta, "y ¿puedo yo...?" "¿Qué tal un café?", le dice.

En el primer punto de este capítulo, se creó una clase llamada "vehículo" y una subclase de "vehículo" llamada "coche":

```
class vehiculo
{
    public void start()
    {
        System.out.println("Arrancar...");
    }
}

class coche extends vehiculo
{
    public void drive()
    {
        System.out.println("Conducir...");
    }
}
```

Esa jerarquía de clases sólo incluía dos niveles, la superclase y la subclase, pero se puede profundizar más. Digamos, por ejemplo, que tenemos una nueva clase, avión, que es una subclase de vehículo y añade un método llamado *fly*:

```
class avion extends vehiculo
{
    public void fly()
    {
        System.out.println("Volar...");
    }
}
```

Hay todo tipo de aviones, y derivará dos clases de "avión": *whirlybird*, que define un nuevo método llamado *whirl*, y *jet*, que define un método llamado *zoom*:

```
class whirlybird extends avion
{
    public void whirl()
    {
        System.out.println("Girar...");
    }
}

class jet extends avion
{
    public void zoom()
    {
        System.out.println("Subir verticalmente...");
    }
}
```

Esta es la jerarquía de la clase ahora:

```
vehiculo
    |_____coche
```

```
    |_____avion
          |_____whirlybird
          |_____jet
```

Ahora podemos instanciar objetos de estas clases, como se hace en este ejemplo en el que se crean objetos de la clase "coche" y *jet*:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un coche...");
        coche c = new coche();
        c.start();
        c.drive();

        System.out.println();
        System.out.println("Crear un jet...");
        jet j = new jet();
        j.start();
        j.fly();
        j.zoom();
    }
}
```

Esta es la salida del código:

```
C:\>java app
Crear un coche...
Arrancar...
Conducir...

Crear un jet...
Arrancar...
Volar...
Subir verticalmente...
```

Gestionar multiniveles de constructores

"¡Eh!" dice el programador novato. "¿Está en plan gracioso Java otra vez?" le pregunta. "Sí", dice el PN, "he creado cuatro niveles de subclases, clases de la a a la d, donde a es la superclase de la que deriva todo. Tiene sentido pensar que el constructor de d será al que se llame primero, luego al de c, luego al de b y después al de a, ¿correcto?" "No", le dice, "Java los llama exactamente en orden inverso". "Sabía que estaría de mi lado", dice el PN.

Miremos en un ejemplo el uso de los multiniveles de constructores. Aquí, implementaremos el programa del programador novato con cuatro niveles de subclasificación, empezando con la clase a:

```
class a
{
    a()
    {
        System.out.println("Construcción de  a...");
    }
}

class b extends a
{
    b()
    {
        System.out.println("Construcción de b...");
    }
}

class c extends b
{
    c()
    {
        System.out.println("Construcción de c...");
    }
}

class d extends c
{
    d()
    {
        System.out.println("Construcción de d...");
    }
}
```

Después, se creará un objeto de la clase d, la última clase de la cadena de subclases:

```
public class app
{
    public static void main(String[] args)
    {
        d obj = new d();
    }
}
```

Esto es lo que se ve cuando se ejecuta el código:

```
C:\>java app
Construcción de a...
Construcción de b...
Construcción de c...
Construcción de d...
```

En otras palabras, llamó primero al constructor de a, después al de b, luego al de c y después al de d, no en el orden inverso como se podía esperar. ¿Por qué Java lo hace de esta forma? Porque cuando se crean subclases, se

procede desde lo general a lo específico, lo que significa que la clase a no sabe nada de la clase b, ésta no sabe nada de la c y así sucesivamente. Por esa razón, Java llama al constructor de la subclase original primero, después al siguiente, etc. Como la clase b sabe de la clase a, se podría contar con tener ciertas partes de a inicializadas antes de completar su inicialización, y lo mismo para la clase c respecto a la b, etc.

Además, observe que se pueden pasar parámetros a los múltiples niveles de las formas que se vieron en el punto correspondiente en este capítulo. Sin embargo, todos los constructores de la cadena de subclases deben llamarse en orden ascendente.

Sobrescritura de métodos

"Bien", dice el programador novato, "creía que podía usar la clase *Button* de Java en mi nuevo programa, pero quería crear un método llamado *getLabel*, y la clase *Button* ya tiene un método con ese nombre". "No hay problema", le dice, "puede sobrescribir el método *getLabel* de la clase *Button* con una nueva implementación de ese método". "¿Puedo hacer eso?" pregunta PN.

En el último capítulo, vimos que se puede sobrecargar métodos con diferentes implementaciones que tienen diferentes listas de parámetros. También se puede sobrecargar métodos que se heredan de una superclase, lo que quiere decir que los sustituye con una nueva versión.

En el ejemplo siguiente, empezaremos con una clase base general llamada animal que tiene un método, *breathe*. Cuando se llama a *breathe*, se visualiza "Respirar..." Este es el código:

```
class animal
{
    public void breathe()
    {
        System.out.println("Respirar...");
    }
}
```

Ahora, supongamos que se quiere derivar una nueva clase de "animal" llamada "pez". Cuando se prueba el método *breathe* en la clase "pez", sin embargo, se ve que se visualiza "Respirar...". Se decide que sería mejor visualizar "Burbujear...". Para hacerlo, se puede sobrescribir el método *breathe* en la clase "pez" simplemente definiendo una nueva versión con la misma lista de parámetros:

```
class animal
{
    public void breathe()
    {
        System.out.println("Respirar...");
    }
}

class pez extends animal
{
    public void breathe()
    {
         System.out.println("Burbujear…");
    }
}
```

Ahora se puede instanciar nuevos objetos de las clases "animal" y "pez" y llamar a sus métodos *breathe* como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un animal...");
        animal a = new animal();
        a.breathe();

        System.out.println();
        System.out.println("Crear un pez...");
        pez f = new pez();
        f.breathe();
    }
}
```

Esta es la salida del código, mostrando que el método *breathe* está sobrecargado:

```
C:\>java app
Crear un animal...
Respirar...

Crear un pez...
Burbujear...
```

Acceso a los miembros sobrescritos

"Bien", dice el programador novato, "creo que he encontrado un problema que nadie más ha encontrado en Java". "Ah ¿sí?", le pregunta. "Sí", dice PN, "he sobrescrito un método de una superclase, y funciona bien la mayor parte del tiempo, pero algunas veces necesito acceder al método original sobrescri-

to". "Es un problema común", le dice, "y se puede resolver con la palabra clave *super*".

Se puede usar *super* igual que se usa la palabra clave *this*, excepto que *super* no se refiere al objeto actual, sino a su superclase. Por ejemplo, echemos un vistazo al código del punto anterior en el que la clase "pez" es una subclase de la clase "animal" y sobrescribe el método *breathe* para que se visualice "Burbujear..." en lugar de "Respirar...":

```
class animal
{
    public void breathe()
    {
        System.out.println("Respirar...");
    }
}

class pez extends animal
{
    public void breathe()
    {
        System.out.println("Burbujear...");
    }
}
```

Ahora, sin embargo, supongamos que se da cuenta de que cierto tipo de peces pueden respirar como los animales, por lo que se añade un nuevo método a la clase pez, *newbreathe*. En este método, le gustaría utilizar el método *breathe* de la superclase, y se puede hacer eso con la palabra clave *super*, como sigue:

```
class animal
{
    public void breathe()
    {
        System.out.println("Respirar...");
    }
}

class pez extends animal
{
    public void breathe()
    {
        System.out.println("Burbujear...");
    }

    public void newbreathe()
    {
        super.breathe();
    }
}
```

Ahora se puede instanciar objetos de las clases "animal" y "pez" y usar el método *newbreathe*, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un animal...");
        animal a = new animal();
        a.breathe();

        System.out.println();
        System.out.println("Crear un pez...");
        pez lf = new pez();
        lf.newbreathe();
    }
}
```

Este es el resultado del código:

```
C:\>java app
Crear un animal...
Respirar...

Crear un pez...
Respirar...
```

Usar variables de superclase con objetos de subclases

El zar de la Programación Exacta (ZPE) aparece y dice, "En C++, se puede asignar una referencia a un objeto de una subclase a un tipo de variable de una superclase. ¿Se puede hacer eso en Java?" El programador novato dice "¿Lo hace por mí?" "Sí", le responde el ZPE.

Un aspecto interesante de la programación orientada a objetos (POO) en Java, que pondremos a funcionar en el siguiente punto, es que se puede asignar una referencia a un objeto de subclase a un tipo de variable de una superclase. Digamos, por ejemplo, que la clase a es la superclase de b y tiene una variable de la clase a. Esto significa que se puede asignar referencias de objetos de clase b a esa variable, así como referencia a objetos de la clase a.

Veamos un ejemplo. Aquí usaremos el multinivel de la jerarquía de clases que ya vimos en este capítulo:

```
class vehiculo
{
    public void start()
```

```
    {
        System.out.println("Arrancar...");
    }
}

class coche extends vehiculo
{
    public void drive()
    {
        System.out.println("Conducir...");
    }
}

class avion extends vehiculo
{
    public void fly()
    {
        System.out.println("Volar...");
    }
}

class whirlybird extends avion
{
    public void whirl()
    {
        System.out.println("Girar...");
    }
}

class jet extends avion
{
    public void zoom()
    {
        System.out.println("Subir verticalmente...");
    }
}
```

Por ejemplo, para crear nuevos objetos de las clases "coche" y *jet,* se puede usar este código:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un coche...");
        car c = new car();
        c.start();
        c.drive();

        System.out.println();
        System.out.println("Crear un jet...");
        jet j = new jet();
        j.start();
    }
}
```

Sin embargo, también se puede asignar el nuevo objeto *jet* a la variable de clase vehiculo, como sigue:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un coche...");
        coche c = new coche();
        c.start();
        c.drive();

        System.out.println();
        System.out.println("Crear un jet...");
        vehiculo j = new jet();
        j.start();
        //j.fly();
        //j.zoom();
    }
}
```

Esta es la salida del código:

```
C:\>java app
Crear un coche...
Arrancar...
Conducir...

Crear un jet...
Arrancar...
```

Ahora se comentaron las líneas *j.fly()* y *j.zoom()* porque esos métodos están definidos en las clases "avión" y *jet*, que son subclases de "vehículo", lo que quiere decir que esos métodos no se pueden usar con una variable de la clase "vehículo". En general, una variable de objeto, a, sólo permitirá acceder a los elementos de su propia clase, no necesariamente a todos los miembros; en particular, no se podrá acceder a ningún miembro que no lo sea de la clase a.

Dispatching dinámico de métodos (Polimorfismo en tiempo de ejecución)

"Vaya", dice el programador novato, "tengo otro problema. Mi programa de dibujo crea objetos de las clases triángulo, cuadrado y círculo, cada una de ellas tiene un método Dibujar, pero no estoy seguro del tipo de objeto que el usuario querrá hacer hasta que el programa no se ejecute. ¿Tendré que escribir el código principal del programa tres veces, una por cada tipo de objeto?"

"No del todo", le dice. "Puede usar el polimorfismo en tiempo de ejecución". "¿Cómo?" contesta PN.

El polimorfismo en tiempo de ejecución, llamado, en Java, *dispatch* dinámico de métodos, permite esperar hasta que el programa se esté realmente ejecutando para especificar el tipo de objeto que estará en una variable particular del objeto. Esto quiere decir que, en el caso del programador novato, puede escribir su código llamando al método Dibujar en varias variables y decidir el tipo de objeto, triángulo, cuadrado o círculo, que está almacenado en esas variables en tiempo de ejecución.

Como vimos en la sección anterior, se puede asignar una referencia a un objeto de subclase a un tipo de variable de una superclase. Puede que se esté preguntando porqué Java, que es tan estricto con el tipo de datos, permite hacer esto. La respuesta es soportar el polimorfismo en tiempo de ejecución.

En el siguiente ejemplo se clarifica todo esto. En este caso, se creará una superclase llamada a, una subclase de a llamada b, una subclase de b llamada c y una subclase de c llamada d, cada una de las cuales tiene un método *print*:

```
class a
{
    public void print()
    {
        System.out.println("Aquí está a...");
    }
}

class b extends a
{
    public void print()
    {
        System.out.println("Aquí está b...");
    }
}

class c extends a
{
    public void print()
    {
        System.out.println("Aquí está c...");
    }
}

class d extends a
{
    public void print()
    {
        System.out.println("Aquí está d...");
    }
}
```

Ahora se puede crear una referencia al objeto de cada tipo de clase:

```
public class app
{
    public static void main(String[] args)
    {
        a a1 = new a();
        b b1 = new b();
        c c1 = new c();
        d d1 = new d();
        .
        .
        .
```

Para mostrar cómo funciona el polimorfismo en tiempo de ejecución, crearemos además una variable llamada *aref* que gestiona una referencia a un objeto de la clase a:

```
public class app
{
    public static void main(String[] args)
    {
        a a1 = new a();
        b b1 = new b();
        c c1 = new c();
        d d1 = new d();
        a aref;
        .
        .
        .
```

Ahora, se puede hacer referencia a los objetos de todas las clases diferentes en *aref*, y cuando se llame al método *print*, el método *print* de la clase correspondiente será el llamado:

```
public class app
{
    public static void main(String[] args)
    {
        a a1 = new a();
        b b1 = new b();
        c c1 = new c();
        d d1 = new d();
        a aref;

        aref = a1;
        aref.print();

        aref = b1;
        aref.print();

        aref = c1;
```

```
            aref.print();

            aref = d1;
            aref.print();
        }
    }
```

Este es el resultado del código:

```
C:\>java app
Aquí está a...
Aquí está b...
Aquí está c...
Aquí está d...
```

Usando el polimorfismo en tiempo de ejecución, se puede escribir código que funcionará con diferentes tipos de objetos y se decidirá el tipo de objeto actual en tiempo de ejecución. Observe que todavía se aplican las restricciones mencionadas en el punto anterior: una variable de objeto, a, sólo permitirá acceder a los elementos que son miembros de su propia clase, no necesariamente a todos los miembros de la variable objeto que gestiona, en particular, no se podrá acceder a los miembros que no son miembros de la clase de a.

Crear clases abstractas

El programador novato aparece. "¡Ese Johnson!", dice PN. "¿Qué ocurre?", le pregunta. "Ese Johnson estaba utilizando una de mis clases que es necesario personalizar antes de que se pueda usar, pero no lo hizo. Por lo tanto, no funcionó, ¡justo allí, delante del gran jefe!" "Bien", le dice, "la próxima vez, haga una clase abstracta, y ese Johnson tendrá que personalizarla".

Cuando se escriben clases, se pueden ejecutar casos donde sólo se pueda proporcionar código general, y es necesario que el desarrollador que crea subclase de la clase las personalice. Para asegurarse de que el desarrollador personalice el código, se puede hacer que el método sea abstracto, lo que significa que el desarrollador tendrá que sobrescribir el método, de lo contrario Java se quejará. Para hacer un método abstracto, se usa la palabra clave *abstract*. Si se hace algún método abstracto en una clase, también deberá serlo la clase.

He aquí un ejemplo. En este caso, se crea una clase llamada a que tiene un método llamado *print*, que visualiza una cadena y obtiene la cadena para visualizar llamando al método *getData*:

```
class a
{
```

```
    String getData();

    public void print()
    {
        System.out.println(getData());
    }
}
```

Observe que no hay ninguna implementación del método *getData* porque se quiere que los desarrolladores especifiquen qué datos van a visualizar. Para asegurarse de que saben que deben proporcionar una implementación del método *getData*, se puede hacer que sea abstracto, lo que significa que también la clase debe ser abstracta:

```
abstract class a
{
    abstract String getData();

    public void print()
    {
        System.out.println(getData());
    }
}
```

Ahora, cuando se crea una subclase de a, hay que proporcionar una implementación de *getData*, como sigue:

```
class b extends a
{
    String getData()
    {
        return "¡Hola desde Java!";
    }
}
```

Así es como funcionaría la subclase (observe que no se puede instanciar directamente un abstracto):

```
public class app
{
    public static void main(String[] args)
    {
        b b1 = new b();

        b1.print();
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola desde Java!
```



Truco: es importante entender está técnica porque una gran cantidad de métodos de los paquetes de Java son abstractos, y por lo tanto deben sobrescribirse.

Abandonar la sobrescritura con *final*

El programador novato dice, "¡Ese maldito Johnson!" "¿Qué ocurre?", le pregunta. "Ese Johnson sobrescribió el método dibujar de mi clase *painter* y lo lió todo", se queja PN. "No se preocupe, PN", le dice, "puede marcar ese método como *final*, y nadie puede sobrescribirlo". "¡Bien!", dice PN.

Anteriormente en este capítulo, vimos un ejemplo en el que la clase "pez" sobrescribía el método *breathe* de la clase "animal":

```
class animal
{
    void breathe()
    {
        System.out.println("Respirar...");
    }
}

class pez extends animal
{
    public void breathe()
    {
        System.out.println("Burbujear...");
    }
}
```

Si por alguna razón no se quiere permitir la sobrescritura del método *breathe*, se le puede declarar final, como sigue:

```
class animal
{
    final void breathe()
    {
        System.out.println("Respirar...");
    }
}

class pez extends animal
{
    public void breathe()
    {
        System.out.println("Burbujear...");
    }
}
```

Ahora, supongamos que se intenta usar estas clases en algún código:

```
public class app
{
    public static void main(String[] args)
    {
        System.out.println("Crear un animal...");
        animal a = new animal();
        a.breathe();

        System.out.println();
        System.out.println("Crear un pez...");
        pez f = new pez();
        f.breathe();
    }
}
```

Java avisará de que no se puede sobrescribir *breathe*, de la siguiente forma:

```
C:\>javac app.java -deprecation
app.java:11: The method void breathe() declared in class pez cannot
override the final method of the same signature declared in class animal.
Final methods cannot be overridden.
    public void breathe()
                ^
1 error
```

Abandonar la herencia con *final*

Se puede prevenir la creación de subclases de una clase declarando toda la clase como *final*, como se ve a continuación:

```
final class animal
{
    public void breathe()
    {
        System.out.println("Respirar...");
    }
}

class pez extends animal
{
    public void breathe()
    {
        System.out.println("Burbujear...");
    }
}

public class app
{
    public static void main(String[] args)
```

```
    {
        System.out.println("Crear un animal...");
        animal a = new animal();
        a.breathe();

        System.out.println();
        System.out.println("Crear un pez...");
        pez f = new pez();
        f.breathe();
    }
}
```

Esto es lo que ocurre cuando se intenta ejecutar este código:

```
C:\>javac app.java -deprecation
app.java:9: Can't subclass final classes: class animal
class pez extends animal
                  ^
1 error
```

Crear constantes con *final*

En los dos puntos anteriores, se mostró dos formas de usar *final*: para prevenir que un método sea sobrescrito y para crear subclases de una clase. En Java, hay otro uso de *final*: se puede usar para declarar constantes.

Este es un ejemplo en el que se convierte una variable en constante con *final* y después se le intenta asignar un valor:

```
public class app
{
    public static void main(String[] args)
    {
        final int a = 5;

        a = 6;
    }
}
```

Esto es lo que el compilador de Java diría:

```
C:\> javac app.java -deprecation
app.java:7: Can't assign a value to a final variable: a
        a = 6;
        ^
1 error
```

Relación es-a frente a tiene-a

El zar de la Programación Exacta llega y dice, "En C++, en la herencia se pueden tener relaciones es-a y tiene-a". "También en Java", le dice.

Se pueden ver los términos es-a y tiene-a al trabajar con la herencia, porque especifican dos de las formas en que las clases pueden relacionarse. La herencia estándar es en la que generalmente se piensa en términos de una relación es-a, como en el ejemplo siguiente. En este caso, la clase a extiende la clase b, por lo que se puede decir que la clase a es-a b:

```
class a extends b
{
    a()
    {
        print();
    }
}

class b
{
    void print()
    {
        System.out.println("Esto viene de la clase b...");
    }
}

public class app
{
    public static void main(String[] args)
    {
        a obj = new a();
    }
}
```

Cuando se llama al método *print* de a, realmente se está llamando al método *print* de a heredado de b, y es este método quien hace la visualización:

```
C:\>java app
Esto viene de la clase b...
```

Por otro lado, en una relación tiene-a, un objeto incluye una referencia a otro, como en este caso, en el que los objetos de la clase a incluirán un objeto interno de la clase b:

```
class a
{
    b b1;

    a()
    {
        b1 = new b();
        b1.print();
    }
}

class b
```

```
{
    void print()
    {
        System.out.println("Esto viene de la clase b...");
    }
}

public class app
{
    public static void main(String[] args)
    {
        a obj = new a();
    }
}
```

Ahora, el método *print* de la clase b está accesible desde el objeto b1 del objeto de la clase a:

```
C:\>java app
Esto viene de la clase b...
```

La clase *Object* de Java

"Lo que quiero", dice el programador novato, "es escribir una clase sencilla, sin herencia ni nada y..." "Demasiado tarde", le dice. "En Java, todas las clases son subclases de una clase maestra, *java.lang.Object*, por lo que en todo está involucrada la herencia".

En Java, toda clase se deriva automáticamente de la clase *java.lang.Object*, y hay ciertas ventajas de saber esto, ya que significa que todos los objetos ya han heredado algunos métodos que están preparados para usarlos. Los métodos de la clase *Object* aparecen en la tabla 5.2.

Este es un ejemplo en el que se usa el método *getClass* para determinar la clase de una referencia a un objeto en una variable de superclase; esto es útil porque una variable de superclase puede gestionar referencias a objetos de cualquiera de sus subclases.

Empezamos con una superclase llamada a y tres subclases: b, c y d. El método *print* de cada una de estas clases visualiza el nombre de la clase. Este es el código:

```
class a
{
    public void print()
    {
        System.out.println("Aquí está a...");
    }
}
```

Tabla 5.2. Los métodos de la clase *Object* de Java.

Constructores	Descripción
protected Object clone()	Proporciona una copia del objeto.
boolean equals(Object obj)	Indica si otro objeto es igual a éste.
Protected void finalize()	El recolector de basura lo llama en un objeto, cuando la colección *garbage* se va a deshacer del objeto.
Class getClass()	Proporciona la clase de un objeto en tiempo de ejecución.
int hashCode()	Proporciona un valor *hashcode* para el objeto.
void notify()	Activa un hilo *(thread)* sencillo que está esperando en este objeto.
void notifyAll()	Activa todos los hilos *(threads)* que están esperando en este objeto.
String toString()	Proporciona una representación en cadena del objeto.
void wait()	Hace que el hilo *(thread)* actual espere hasta que otro invoque el método *notify* o el *notifyAll.*
void wait(long timeout)	Hace que el hilo *(thread)* actual espere hasta que otro invoque el método *notify* o el *notifyAll* o pase una cierta cantidad de tiempo.
void wait(long timeout, int nanos)	Hace que el hilo *(thread)* actual espere hasta que otro invoque el método *notify* o el *notifyAll* para este objeto, algún otro hilo interrumpa a éste o pase una cierta cantidad de tiempo.

```
class b extends a
{
    public void print()
    {
        System.out.println("Aquí está b...");
    }
}

class c extends a
{
    public void print()
    {
        System.out.println("Aquí está c...");
    }
```

```
}

class d extends a
{
    public void print()
    {
        System.out.println("Aquí está d...");
    }
}
```

Lo siguiente, es crear una instancia de cada clase y una variable de la clase a llamada *aref:*

```
public class app
{
    public static void main(String[] args)
    {
        a a1 = new a();
        b b1 = new b();
        c c1 = new c();
        d d1 = new d();
        a aref;
        .
        .
        .
```

Ahora, se puede determinar la clase del objeto en *aref*, no importa cuál sea la subclase:

```
public class app
{
    public static void main(String[] args)
    {
        a a1 = new a();
        b b1 = new b();
        c c1 = new c();
        d d1 = new d();
        a aref;

        aref = a1;
     System.out.println("Ahora, la clase de aref es " + aref.getClass());
        Aref.print();

        aref = b1;
     System.out.println("Ahora, la clase de aref es " + aref.getClass());
        Aref.print();

        aref = c1;
     System.out.println("Ahora, la clase de aref es " + aref.getClass());
        Aref.print();

        aref = d1;
     System.out.println("Ahora, la clase de aref es " + aref.getClass());
        Aref.print();
```

```
    }
}
```

Este es el resultado:

```
C:\>java app
Ahora, la clase de aref es a
Aquí está a...
Ahora, la clase de aref es b
Aquí está b...
Ahora, la clase de aref es c
Aquí está c...
Ahora, la clase de aref es d
Aquí está d...
```

Como se puede observar, los métodos construidos en cada objeto, como *getClass*, pueden ser muy útiles.

Usar interfaces para herencia múltiple

"Hmm", dice el programador novato, "tengo una clase llamada "animal" y una clase totalmente separada llamada "mineral" y quiero heredar de ambas clases al mismo tiempo para crear algo totalmente nuevo". "Lo siento, " le dice, "Java no soporta la herencia múltiple directamente, tendrá que hacer que una de esas clases sea una interfaz".

En otros lenguajes, como C++, una clase puede heredar de múltiples clases de una vez, pero esta técnica no funciona en Java directamente, es decir, sólo se puede usar, de una vez, la palabra clave *extends* con una clase. Por lo tanto, no se puede hacer esto:

```
class a extends b, c      //¡No funcionará!
{
    .
    .
    .
}
```

Sin embargo, hay dos formas de implementar la herencia múltiple en Java. La primera es usar herencia sencilla por etapas (funcionará para las clases de las que se quiere heredar), como sigue:

```
class c
{
    .
    .
    .
```

```
}

class b extends c
{
    .
    .
    .
}

class a extends b
{
    .
    .
    .
}
```

La otra manera es usar interfaces. Las interfaces empezarán a ser importantes en el siguiente capítulo, por lo tanto, aquí sólo haremos una introducción.

Una interfaz especifica la forma de sus métodos pero no da ningún detalle de su implementación; por lo que se puede pensar en ella como la declaración de una clase. Como veremos más tarde en el libro, se pueden crear interfaces con la palabra clave *interface*:

```
interface c
{
    .
    .
    .
}

interface b
{
    .
    .
    .
}
```

Ahora, se pueden usar estas dos interfaces con la palabra clave *implements*:

```
class a implements b, c
{
    .
    .
    .
}
```

Dado que las interfaces declaran pero no definen métodos, es labor de cada uno implementar los métodos de las interfaces. Por ejemplo, en la *applet* que se mostró al principio del capítulo, se implementó la interfaz de Java *ActionListener* para gestionar los clics de los botones (se verán todos los

detalles de las *applets* en el siguiente capítulo). Esa interfaz declara un método, *actiopnerformed*, que se define como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class clicker extends Applet implements ActionListener {

     TextField text1;
     Button button1;

     public void init(){
               text1 = new TextField(20);
               add(text1);
               button1 = new Button("¡Hacer clic aquí!");
               add(button1);
          button1.addActionListener(this);
       }

       public void actionPerformed(ActionEvent event){
                String msg = new String("Bienvenido a Java");
                if(event.getSource() == button1){
                        Text1.setText(msg);
                }
        }
}
```

Si no se define el método *actionPerformed*, el compilador de Java dará un mensaje como este:

```
C:\>javac clicker.java -deprecation
clicker.java:5: class clicker must be declared abstract. It does not
define
void actionPerformed(java.awt.event.ActionEvent) from interface
java.awt.event.ActionListener.
public class clicker extends Applet implements ActionListener {
             ^
1 error
```

Se pueden implementar tantas interfaces como se quiera, como en este ejemplo, en el que se implementa la interfaz *ActionListener* para los clics de los botones y las interfaces *MouseListener* y *MouseMotionListener* para trabajar con el ratón:

```
import java.awt.Graphics;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class dauber extends Applet implements ActionListener,
    MouseListener, MouseMotionListener {

        Button buttonDraw, buttonLine, buttonOval, buttonRect,
```

```
    button3DRect;
Button buttonRounded;

Point pts[] = new Point[1000];
Point ptAnchor, ptDrawTo, ptOldAnchor, ptOldDrawTo;
int ptindex = 0;
    .
    .
    .
```

Crear clases internas

"De acuerdo", dice el programador novato, "ahora soy un experto. ¿Hay algo más sobre clases que no sepa?" Usted sonríe y dice, "Bastante. Por ejemplo, ¿qué sabe de las clases internas?" "¿Qué clases?", pregunta PN.

Al empezar en 1.1, se podían anidar definiciones de clases unas dentro de otras. Las clases anidadas pueden ser estáticas o no estáticas; sin embargo, las clases estáticas no pueden hacer referencia a los miembros de las clases que tiene en su interior directamente, sino que deben instanciar y usar un objeto en su lugar, por lo que con frecuencia no se usan. Las clases internas, por otro lado, son clases anidadas no estáticas, y son bastante más populares por ser útiles en la gestión de eventos, como veremos en el siguiente capítulo.

Esto es un ejemplo de clases internas; en esta clase, la clase b está definida en el interior de la clase a y se instancia un objeto de la clase b en el constructor de la clase a:

```
class a
{
    b obj;

    a()
    {
        obj = new b();
        obj.print();
    }

    class b
    {
         public void print()
         {
             System.out.println("En el interior de b…");
         }
    }
}

public class app
{
```

```
    public static void main(String[] args)
    {
        a obj = new a();
    }
}
```

Cuando se ejecuta este código, se instancia un objeto de la clase a, que instancia un objeto interno de la clase b y llama al método *print* de ese objeto. Este es el resultado:

```
C:\>java app
En el interior de b...
```

Más allá de las clases internas, como se ha visto en este ejemplo, se pueden tener clases internas anónimas (sin nombre) (ver el siguiente punto para más detalles).

Crear clases internas anónimas

Un forma corta que es útil para trabajar con la gestión de eventos, es usar clases internas anónimas. Las vamos a introducir aquí y las usaremos en el siguiente capítulo. Una clase interna anónima es aquella que no tiene nombre y se crea usando la siguiente sintaxis:

```
new SuperType(constructor parameters)
{
    //métodos y datos
}
```

Aquí, *SuperType* es el nombre de una clase o interfaz de la que se están creando subclases (se debe especificar un tipo de superclase cuando se crean clases internas anónimas), y se puede definir los métodos y los datos de las clases internas anónimas en un bloque de código.

Veamos un ejemplo. En este caso, crearemos una clase interna anónima en el interior de una nueva clase, clase a. Esta clase interna anónima subclasificará otra clase, clase b y definirá un método llamado *print*, al que se llamará inmediatamente:

```
class a
{
    a()
    {
        (new b() {public void print()
            {System.out.println("¡Hola desde Java!");}}).print();
    }
}
```

```
class b{}
```

Lo único que queda es crear un objeto de la clase a y llamar al constructor de esa clase:

```
public class app
{
    public static void main(String[] args)
    {
        a obj = new a();
    }
}
```

Este es el resultado del código:

```
C:\>java app
¡Hola desde Java!
```

En el siguiente capítulo, aprenderemos más sobre las clases internas anónimas.

6 AWT: Applets, aplicaciones y gestión de eventos

Hemos trabajado mucho con la sintaxis de Java para llegar a este punto, que es uno de los capítulos principales. Aquí empezaremos a trabajar con los programas de gráficos, las *applets* y las aplicaciones. Este capítulo introduce *Java Abstract Windowing Toolkit* (AWT), la forma original de Java para trabajar con gráficos.

Ahora, AWT está complementado con el paquete *Swing*, que veremos dentro de unos cuantos capítulos. AWT proporciona la base para trabajar con gráficos en Java e incluso el paquete *Swing* está basado en él.

En este capítulo, trabajaremos con AWT, creando *applets* y ventanas de aplicación. Antes de empezar, situaremos AWT en la historia. AWT fue desarrollado muy rápidamente para la primera *release* de Java, de hecho, en sólo seis semanas. Los desarrolladores del AWT original usaron una ventana para cada uno de sus componentes, es decir, para cada botón, cuadros de texto, casilla de activación, etc., y por lo tanto, tenían su propia ventana cuando al sistema operativo le interesaba. Esto producía un consumo considerable de los recursos del sistema cuando los programas llegaban a ser grandes. Recientemente, *Sun* introdujo el paquete *Swing*, en el que los componentes se visualizan usando métodos gráficos de la *applet* o aplicación que los contiene, ellos no tienen sus propias ventanas del sistema operativo. Los componentes AWT se llaman componentes pesos pesados debido al uso

significativo que hacen de los recursos del sistema y los componentes *Swing* se llaman componentes peso ligero, ya que no necesitan sus propias ventanas.

¿Qué significa esto? Está claro que para *Sun*, *Swing* es el futuro. Hay más componentes *Swing* que AWT, y de hecho, hay un componente *Swing* para cada componente AWT. En el futuro, probablemente, *Sun* no ampliará el conjunto de componentes AWT mucho más, mientras que sí se espera que *Swing* crezca. Por otro lado, *Swing*, en sí mismo, está basado en AWT; las ventanas que *Swing* utiliza para visualizar sus componentes, es decir, ventanas, marcos de ventanas, *applets* y diálogos, están todos ellos basados en contenedores AWT. AWT no va a desaparecer y para trabajar con *Swing*, se necesita AWT. Por esa razón, y porque gran parte del desarrollo está hecho con AWT, y más que habrá en el futuro, dedicaremos éste y los siguientes capítulos a AWT. Empezaremos con una visión de AWT en sí mismo.

Abstract Windowing Toolkit

No es una exageración decir que la aparición de *Abstract Windowing Toolkit* fue obligada tras la popularidad de Java. Se puede crear y visualizar botones, etiquetas, menús, cuadros de lista desplegables, cuadros de texto y otros controles de la interfaz de usuario que se esperan utilizar en la programación basada en ventanas utilizando AWT. Esta es una lista de las clases de AWT más populares:

- *Applet*: Crea una *applet*.
- *Button*: Crea un botón.
- *Canvas*: Crea un área de trabajo en el que se puede dibujar.
- *Checkbox*: Crea una casilla de activación.
- *Choice*: Crea un control de opción.
- *Label*: Crea una etiqueta.
- *Menu*: Crea un menú.
- *ComboBox*: Crea un cuadro de lista desplegable.
- *List*: Crea un cuadro de lista.
- *Frame*: Crea un marco para las ventanas de aplicación.
- *Dialog*: Crea un cuadro de diálogo.
- *Panel*: Crea un área de trabajo que puede tener otros controles.

- *PopupMenu*: Crea un menú emergente.
- *RadioButton*: Crea un botón de opción.
- *ScrollBar*: Crea una barra de desplazamiento.
- *ScrollPane*: Crea un cuadro de desplazamiento.
- *TextArea*: Crea un área de texto de dos dimensiones.
- *TextField*: Crea un cuadro de texto de una dimensión (en otros lenguajes se llama *TextBox*).
- *TextPane*: Crea un área de texto.
- *Window*: Crea una ventana .

La clase *Applet* de AWT es aquella en la que están basadas las *applets* AWT. Primero veremos esta clase.

Applets

¿Qué es exactamente una *applet* AWT? Una *applet* es un fichero de clase que se escribe específicamente para visualizar gráficos en la red Internet. Las *applets* se incluyen en las páginas Web utilizando la etiqueta HTML <APPLET>. Cuando se ejecutan en una página Web, las *applets* de Java se descargan automáticamente y el *browser* las ejecuta, visualizándose en el espacio de la página que se ha reservado para ellas. Pueden hacer de todo, desde trabajar con gráficos hasta visualizar animaciones, gestionar controles (como los que veremos funcionar en este capítulo), cuadros de texto y botones. El uso de las *applets* hace que las páginas Web sean activas, no pasivas, que es su principal atracción.

Cuando se trabaja con AWT, el proceso es como sigue: se crea una *applet* nueva, basándola en la clase *java.applet.Applet* que, a su vez, está basada en la clase *Component* de AWT. He aquí un ejemplo que hemos visto antes y que haremos de nuevo en este capítulo. Este ejemplo visualiza el texto "¡Hola desde Java!" en una página Web:

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}
```

La *applet* se compila en un fichero de *bytecode* con extensión "*.class*". Una vez que se tiene este fichero, se sube a un proveedor de servicios de la red Internet (ISP).

A una *applet* se le puede dar la misma protección que se daría a una página Web, asegurándose de que cualquiera pueda leer el fichero de la *applet* con extensión "*.class*" (por ejemplo, en Unix, se debería dar a la *applet* el permiso 644, que permite a todo el mundo leer el fichero).



Truco: los permisos de los ficheros Unix se corresponden con tres dígitos en formato octal, ordenados, siendo cada uno de ellos el permiso del propietario del fichero, los permisos de otros del mismo grupo de usuarios y los permisos del resto. Para cada dígito octal, un valor de 4 indica permiso de lectura, 2 indica permiso de escritura y 1 indica permiso de ejecución. Poniendo todos estos valores juntos se forma el parámetro de permiso; por ejemplo, un permiso de 0600 significa que el propietario del fichero, y sólo él, puede leer y escribir el fichero.

La nueva *applet* se puede insertar en una página Web con la etiqueta <APPLET>, indicando el nombre del fichero con extensión "*.class*" así como decirle al *browser* cuánto espacio (en pixels) requiere la *applet*. Esto es un ejemplo:

```
<HTML>

 <BODY>

 <CENTER>
 <APPLET
    CODE = "newApplet.class"
    WIDTH = 300
    HEIGHT = 200
 >
 </APPLET>
 </CENTER>

 </BODY>
 </HTML>
```

En este caso, hemos establecido un espacio centrado de 300 x 200 pixels en una página Web que visualiza la *applet*, y decimos al *browser* que descargue el fichero *newApplet.class* y lo ejecute. En este capítulo, se incluirán los detalles de la etiqueta <APPLET>. Cuando el *browser* abra esta página, se visualizará la *applet*.



Truco: un elemento útil que hay que conocer es el *plug-in* de Java del *browser*. Los fabricantes de los Web *browser* han sido bastante lentos para implementar las últimas versiones de Java, por lo que *Sun* tuvo en sus manos el crear el Java *plug-in*, que es un *plug-in* para *Netscape Navigator* y un control *ActiveX* para *Microsoft Internet Explorer*. El *plug-in* es una implementación completa del entorno de ejecución de Java, y permite ejecutar las *applets* usando la última versión de Java. Observe, sin embargo, que se deben configurar las páginas Web para usar el *plug-in*. En este capítulo se verá el uso de *plug-in*.

Aplicaciones

Las ventanas de aplicación AWT están basadas en la clase *Frame* de AWT, que crea una ventana con un marco que visualiza botones y un título. Este es un ejemplo que se verá más adelante en este capítulo.

Al igual que la *applet* anterior, esta aplicación visualiza "¡Hola desde Java!" en una ventana:

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

class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}

public class app
{
    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new WindowAdapter() { public void
           windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
    }
}
```

En este capítulo se verá, con más detalle, la creación de las ventanas de aplicación.

Uno de los aspectos más importantes de la creación de *applets* y aplicaciones es permitir al usuario interactuar con el programa por medio de los eventos. Cuando el usuario ejecuta alguna acción, (hacer clic sobre un botón, cerrar una ventana, seleccionar un elemento de una lista o usar el ratón, por ejemplo), Java lo considera un evento. Gestionaremos eventos en el resto del libro y echaremos un vistazo al funcionamiento de la gestión de eventos en este capítulo.

Para empezar a trabajar con eventos, introduciremos dos controles básicos en este capítulo: botones y cuadros de texto. El usuario puede utilizar el ratón para hacer clic sobre botones, lo que inicia alguna acción en el programa, como es escribir texto en un cuadro de texto. De hecho, el clic sobre los botones, es quizás el evento más básico que Java soporta. Para ver cómo funciona la gestión de eventos en Java, en este capítulo se crearán programas que soportan botones y cuadros de texto. En este capítulo aparecerán todos los detalles de estos controles.

Ahora que ya se tiene una visión general de las *applets*, aplicaciones y gestión de eventos, ya es hora de entrar en detalles con la siguiente sección.

Usar *Abstract Windowing Toolkit*

"De acuerdo", dice el programador novato, "estoy preparado para trabajar con *Abstract Windowing Toolkit*. ¿Por dónde empezamos?" "Por la clase *Component*", le dice. "Cojamos un café y revisémoslo".

La clase más básica de AWT es la clase *java.awt.Component,* en la que se basan todos los componentes visuales de AWT. Por ejemplo, la clase *Button, java.awt.Button,*se deriva directamente de *java.awt.Component.* La clase *java.awt.Component*, se deriva directamente de la clase *java.lang.Object*, que se vio en el capítulo anterior.

La clase *Component* incluye un gran número de métodos, muchos de ellos se verán en este y en los siguientes capítulos. Todos ellos se han incluido en la tabla 6.1 como referencia. Esta es una tabla larga, pero es bueno tenerla y regresar a ella más tarde como referencia.

Tabla 6.1. Métodos de la clase Component de AWT.

Método	Acción
boolean action(Event evento, *Object* objeto*)*	Obsoleto. Se debería registrar el componente con *ActionListener.*

Método	Acción
void add(PopupMenu menu_emergente*)*	Añade al componente el menú emergente indicado.
void addComponentListener(ComponentListener l)	Añade el *ComponentListener* para recibir los eventos del componente.
void addFocusListener(FocusListener l)	Añade el *FocusListener* para recibir eventos de foco.
void addInputMethodListener(inputMehtodListener l)	Añade el *inputMethodListener* para recibir los eventos del método de entrada.
void addKeyListener(KeyListener l)	Añade *KeyListener* para recibir eventos de clave.
void addMouseListener(MouseListener l)	Añade el *MouseL*istener para recibir eventos de ratón.
void addMouseMotionListener(MouseMotionListener l)	Añade el *MouseMotionListener* para recibir los eventos de movimiento del ratón.
void addNotify()	Añade un componente visualizable para conectarlo a un recurso de pantalla nativo.
void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)	Añade un *PropertyChangeListener.*
Rectangle bounds()	Obsoleto. Sustituido por *getBounds*().
int checkImage(Image image, InmageObserver observer)	Obtiene el estado de la representación en pantalla de la imagen.
protected AWTEvent coalesceEvents (AWTEvent evento_existente, *AWTEvent* nuevo_evento)	Fusiona un evento para ser enviado con otro evento.
boolean contains(int x, int y)	Comprueba si este componente contiene el punto indicado.
boolean contains(Point p)	Comprueba si este componente contiene el punto indicado.

Método	Acción
Image createImage(ImageProducer productor*)*	Crea una imagen desde el productor de imagen indicado.
Image createImage(int anchura, *int* altura*)*	Crea una imagen *offscreen* para usarla como doble *buffer.*
void deliverEvent(Event e)	Obsoleto. Sustituido por *dispatchEvent(AWTEvent e).*
void disable()	Obsoleto. Sustituido por *setEnabled(boolean).*
protected void disableEvents(long eventos_a_deshabilitar*)*	Deshabilita los eventos definidos por el evento indicado como parámetro.
void dispatchEvent(AWTEvent e)	Despacha un evento a este componente o a uno de los subcomponentes.
void doLayout()	Hace que el gestor de esquemas ponga este componente.
void enable()	Obsoleto. Sustituido por *setEnabled(boolean).*
void enable(boolean b)	Obsoleto. Sustituido por *setEnabled(boolean).*
protected void enableEvents(long eventos_a_habilitar*)*	Habilita los eventos definidos por el evento indicado para enviarlo a este componente.
voidenableInputMethods(boolean enable)	Habilita o deshabilita el soporte del método de entrada.
protected void firePropertyChange (String propertyName, Object valor_antiguo, *Object* valor_nuevo*)*	Fija el soporte para reportar los cambios de las propiedades encerradas.
float getAlignmentX()	Alineación a lo largo del eje x.
float getAlignmentY()	Alineación a lo largo del eje y.
Color getBackground()	Obtiene el color del fondo de este componente.
Rectangle getBounds()	Establece los límites de este componente en un objeto rectángulo.

Método	Acción
Rectangle getBounds(Rectangle rv)	Almacena los límites de este componente en rv y devuelve rv.
ColorModel getColorModel()	Obtiene la instancia de *ColorModel* usándolo para visualizar el componente.
Component getComponentAt (int x, int y)	Determina si el componente o uno de sus subcomponentes inmediatos contiene la localización (x, y) y obtiene el componente contenido.
Componente getComponentAt(Point p)	Obtiene el componente o subcomponente que contiene el punto indicado.
ComponentOrientation getComponent Orientation()	Establece la orientación sensible al lenguaje para ordenar los elementos o el texto dentro de este componente.
Cursor getCursor	Obtiene el cursor puesto en el componente.
DropTarget getDropTarget()	Obtiene el *DropTarget* conectado a este componente.
Font getFont()	Obtiene la fuente de este componente.
FontMetrics getFontMetrics(Font fuente*)*	Obtiene la métrica de la fuente indicada.
Color getForegorund()	Obtiene el color de primer plano de este componente.
Graphics getGraphics()	Crea un contexto gráfico para este componente.
int getHeigth()	Devuelve la altura actual de este componente.
InputContext getInputContext()	Obtiene el contexto de entrada usado por este componente.
InputMethodRequests getInputMethod-Requests()	Obtiene el método de entrada solicitado.

Método	Acción
Locale getLocale()	Obtiene el *locale* de este componente.
Point getLocale()	Obtiene la ubicación de este componente en la forma de un punto especificando la esquina superior izquierda del componente.
Point getLocation(Point rv)	Almacena el origen x, y de este componente en rv y devuelve rv.
Point getLocationOnScreen()	Obtiene la localización de este componente en forma de punto, especificando la esquina superior izquierda del componente.
Dimension getMaximumSize()	Obtiene el tamaño máximo de este componente.
Dimension getMinimumSize()	Obtiene el tamaño mínimo de este componente.
String getName()	Obtiene el nombre del componente.
Container getParent()	Obtiene el padre de este componente.
java.awt.peer.ComponentPeer getPeer()	Obsoleto. Sustituido por *boolean isDisplayable().*
dimension getPreferredSize()	Obtiene el tamaño preferido de este componente.
Dimension getSize()	Obtiene el tamaño de este componente de un objeto *Dimension.*
Dimension getSize(Dimension rv)	Almacena el ancho y el alto de este componente en rv y devuelve rv.
Toolkit getToolkit()	Obtiene el kit de herramientas de este componente.
Object getTreeLock()	Obtiene el objeto *locking* para el árbol de componente AWT.
int getWidth()	Obtiene la anchura actual de este componente.

Método	Acción
int getX()	Obtiene la coordenada actual x del componente original.
int getY()	Obtiene la coordenada actual y del componente original.
boolean gotFocus(Event evento, *Object* objeto*)*	Obsoleto. Sustituido por*processFocusEvent(FocusEvent).*
boolean handleEvent(Event evento*)*	Obsoleto. Sustituido por*processEvent(AWTEvent*).
boolean hasFocus()	Verdadero si este componente tiene el foco del teclado.
void hide()	Obsoleto. Reemplazado por*setVisible(boolean).*
boolean imageUpdate(image img, int flags, int x, int y, int w, int h)	Repinta el componente cuando cambia la imagen.
boolean inside(int x, int y)	Obsoleto. Sustituido por*contains (int, int*).
void invalidate()	Invalida el componente.
boolean isDisplayable()	Determina si el componente se puede visualizar.
boolean isDoubleBuffered()	Verdadero si este componente se pinta en una imagen*offscreen* que se copia en la pantalla más tarde.
boolean isEnabled()	Indica si este componente está habilitado.
boolean isFocustraversable()	Indica si el componente puede ser recorrido usando Tab o Shift+ Tab.
boolean isLightweight()	Indica si el componente es peso ligero.
boolean isOpaque()	Verdadero si este componente es completamente opaco, falso por defecto.
boolean isshowing()	Determina si este componente es visible en pantalla.

Método	Acción
boolean isValid()	Determina si el componente es válido.
boolean isVisible()	Determina si el componente debería ser visible cuando el padre lo es.
Boolean keyDown(Event evt, int key)	Obsoleto. Reemplazado por *processKeyEvent(KeyEvent).*
Boolean keyUp(Event evt, int key)	Obsoleto. Reemplazado por *processKeyEvent(KeyEvent).*
Void layout()	Obsoleto. Reemplazado por *doLayout().*
Void list()	Imprime un listado de este componente en el sistema de salida.
Void list(PrintStream out)	Imprime un listado de este componente en el flujo de salida indicado.
Void list(PrintStream out, int indentación*)*	Imprime una lista, empezando en la indentación indicada, en el flujo de salida indicado.
Void list(PrintWriter out)	Imprime un listado en el *PrintWriter* indicado.
Void list(PrintWriter out, int indentación*)*	Imprime una lista, empezando en la indentación indicada, en el *PrintWriter* indicado.
Component locate(int x, int y)	Obsoleto. Reemplazado por *getComponentAt(int, int).*
Point location()	Obsoleto. Reemplazado por *getLocation().*
Boolean lostFocus(Event evento, *Object* objeto*)*	Obsoleto. Reemplazado por *processFocusEvent(FocusEvent).*
Dimension minimumSize()	Obsoleto. Reemplazado por *getMinimumSize().*
Boolean mouseDown(Event evento, *int x, int y)*	Obsoleto. Reemplazado por *processMouseEvent(MouseEvent).*

Método	Acción
Boolean mouseDrag(Event evento, *i nt x, int y)*	Obsoleto. Reemplazado por *processMouseMotionEvent(Mouse Event).*
Boolean mouseEnter(Event evento, *int x, int y)*	Obsoleto. Reemplazado por *processMouseEvent(MouseEvent).*
Boolean mouseExit(Event evento, *int x, int y)*	Obsoleto. Reemplazado por *processMouseEvent(MouseEvent).*
Boolean mouseMove(Event evento, *int s, in ty)*	Obsoleto. Reemplazado por *pro essMouseMotionEvent(Mouse-Event).*
Boolean mouseUp(Event evento, *int x, int y)*	Obsoleto. Reemplazado por *processMouseEvent(MouseEvent).*
Void move(int x, int y)	Obsoleto. Reemplazado por *set-Location(int, int).*
Void nextFocus()	Obsoleto. Reemplazado por *transferFocus().*
Void paint(Graphics g)	Pinta el componente.
Void paintAll(Graphics g)	Pinta el componente y todos los subcomponentes.
Protected String paramString()	Obtiene una cadena que representa el estado del componente.
Boolean postEvent(Event e)	Obsoleto. Reemplazado por *dispatchEvent(AWTEvent).*
Dimension preferredSize()	Obsoleto. Reemplazado por *getPreferredSize().*
Boolean prepareImage(Image image, ImageObserver observer)	Prepara una imagen para enviar a ese componente.
Boolean prepareImage(Image imagen, *int* anchura, *int* altura, *ImageObserver* observador*)*	Prepara una imagen para enviar a ese componente, en la posición indicada.
Void print(Graphics g)	Imprime el componente.
Void printAll(Graphics g)	Imprime el componente y todos sus subcomponentes.
Protected void processComponente-) Event(ComponentEvent e	Procesa los eventos del componente que ocurren en este com-

Método	Acción
	ponente enviándolos a cualquiera de los *ComponenteListeners* registrados.
Protected void processEvent(AWTEvent e)	Procesa los eventos que ocurran en este componente.
Protected void processFocusEvent (FocusEvent e)	Procesa los eventos de foco que ocurran en este componente enviándolos a cualquiera de los objetos *FocusListener* registrados.
Protected void processInputMethodEvent(InputMethodEvent e)	Procesa los eventos del método de entrada que ocurran en este componente enviándolos a cualquiera de los objetos *InputMethodListener* registrados.
Protected void processKeyEvent (KeyEvent e)	Procesa los eventos clave que ocurran en este componente enviándolos a cualquiera de los objetos *KeyListener* registrados.
Protected void processMouseEvent (MouseEvent e)	Procesa los eventos de ratón que ocurran en este componente enviándolos a cualquiera de los objetos *MouseListener* registrados.
Protected void processMouseMotionEvent(MouseEvent e)	Procesa los eventos de movimiento de ratón que ocurran en este componente enviándolos a cualquiera de los objetos *MouseMotionListener* registrados.
Void remove(MenuComponent popup)	Retira del componente el menú emergente indicado.
Void removeComponentListener (ComponentListener l)	Retira el componente *listener* indicado para que no reciba más eventos de componente.
Void removeFocusListener(FocusListener l)	Retira el *focus listener* indicado para que no reciba más eventos de foco.
Void removeInputMethodListener (InputMethodListener l)	Retira el método de entrada indicado para que no reciba más eventos del método *Input.*

Método	Acción
void removeKeyListener(KeyKistener l)	Retira la *key listener* indicada para que no reciba más eventos de este tipo.
void removeMouseListerner(MouseListener l)	Retira el *listener* de ratón indicado para que no reciba más eventos de ratón.
void removeMouseMotionListener (MouseMotionListener l)	Retira el *listener* de movimiento de ratón indicado para que no reciba más eventos de ratón de este tipo.
void removeNotify()	Deshabilita la posibilidad de visualizar el componente, destruyendo su recurso de visualización nativo.
void removePropertyChangeListener (PropertyChangeListener listener)	Retira un *PropertyChangeListener.*
void removePropertyChangeListener (String propertyName, PropertyChange Listener listener)	Retira un *PropertyChangeListener* para una propiedad dada.
void repaint()	Repinta el componente.
void repaint(int x, int y, int anchura, *int* altura*)*	Repinta el rectángulo indicado.
void repaint(long tm)	Repinta el componente.
void repaint(long tm, int x, int y, int anchura, *int* altura*)*	Repinta el rectángulo indicado dentro de tm.
void requestFocus()	Solicita el foco de entrada.
void reshape(int x, int y, int anchura, *int* altura*)*	Obsoleto. Reemplazado por setBounds(int, int, int, int).
void resize(Dimension d)	Obsoleto. Reemplazado por*setSize(Dimension).*
void resize(int anchura, *int* altura*)*	Obsoleto. Reemplazado por*setSize(int, int).*
void setBackground(Color c)	Establece el color de fondo.
void setBounds(int x, int y, int anchura, *int* altura*)*	Mueve y redimensiona el componente.

Método	Acción
void setComponentOrientation(ComponentOrientation o)	Establece la orientación sensible al idioma que se usará para ordenar los elementos o el texto.
void setCursor(Cursor cursor)	Establece la imagen del cursor al cursor indicado.
void setDropTarget(DropTarget dt)	Asocia un *DorpTarget* con este componente.
void setEnabled(boolean b)	Habilita o deshabilita este componente, dependiendo del valor del parámetro .
void setFont(Font f)	Fija la fuente del componente.
void setForeground(Color c)	Fija el color de primer plano del componente.
void setLocale (Locale l)	Fija el *locale* del componente.
void setLocation(int x, int y)	Mueve este componente a una nueva localización.
void setLocation(Point p)	Mueve este componente a una nueva localización.
void setName(String nombre*)*	Fija el nombre del componente a la cadena indicada.
void setSize(Dimension d)	Redimensiona este componente para que tenga la anchura *d.width* y la altura *d.height.*
void setSize(int anchura, *int* altura*)*	Redimensiona este componente para que tenga la anchura y la altura indicadas.
void setVisible(boolean b)	Muestra o esconde este componente tal y como se especifica con el valor del parámetro b.
void show()	Obsoleto. Reemplazado por *setVisible(boolean).*
void show(boolean b)	Obsoleto. Reemplazado por *setVisible(boolean).*
Dimension size()	Obsoleto. Reemplazado por *getSize().*

Método	Acción
String to String()	Obtiene una representación en cadena del componente.
void transferFocus()	Transfiere el foco al siguiente componente.
void update(Graphics g)	Actualiza el componente.
void validate()	Asegura que el componente tiene un esquema válido.

Otra clase importante de AWT es la clase *Container*. Esta clase se deriva de la clase AWT *Component* y es la base de los contenedores AWT, que puede gestionar otros componentes. Las *applets* y las ventanas de aplicación, que pueden visualizar componentes AWT, se basan en la clase *Container*. Dado que en la programación con AWT se utiliza con frecuencia la clase *Container*, como se ve en el siguiente apartado, en la tabla 6.2 aparecen listados sus métodos.

Tabla 6.2. Métodos de la clase *Container*.

Método	Descripción
Component add(Component comp)	Añade el componente indicado al contenedor.
Component add(Component comp, int index)	Añade el componente indicado al contenedor en una posición concreta.
void add(Component comp, Object constraints)	Añade el componente indicado al contenedor.
void add(Component comp, Object constraints, int indice*)*	Añade, en el índice, el componente indicado a este contenedor con las restricciones indicadas.
Component add(String name, Component comp)	Añade un componente a este contenedor.
void addContainerListener(Container Listener l)	Añade un *container listener* para obtener los eventos *container* del contenedor.
protected void addImpl(Component comp, Object constraints, int indice*)*	Añade, en el índice dado, el componente indicado al contenedor.
void addNotify()	Permite visualizar el contenedor conectándolo a un recurso de pantalla nativo.

Método	Descripción
int countComponentes() *(int x, int y)*	Obsoleto. Reemplazado por *getComponentCount()*.
void deliverEvent(Event e)	Obsoleto. Reemplazado por *dispatchEvent(AWTEvent e)*.
void doLayout()	Hace que el contenedor coloque sus componentes.
Component findComponentAt	Localiza el componente hijo visible que contiene la posición indicada.
Component findComponentAt(Point p)	Localiza el componente hijo visible que contiene el punto indicado.
float getAlignmentX()	Obtiene la alineación a lo largo del eje x.
float getAlignmentY()	Obtiene la alineación a lo largo del eje y.
Component getComponent(int n)	Obtiene el componente enésimo del contenedor.
Component getComponentAt(int x, int y)	Localiza el componente que contiene la posición x, y.
Component getComponentAt(Point p)	Obtiene el componente que contiene el punto indicado.
int getComponentCount()	Obtiene el número de componentes en este panel.
Component[] getComponents()	Obtiene todos los componentes del contenedor.
Insets getInsets()	Determina las intercalaciones del contenedor, que indican el tamaño del borde del mismo.
LayoutManager getLayout()	Obtiene el gestor de esquemas del contenedor.
Dimension getMaximumSize()	Obtiene el tamaño máximo del contenedor.
Dimension getMinimumSize()	Obtiene el tamaño mínimo del contenedor.
Dimension getPreferredSize()	Obtiene el tamaño preferido del contenedor.

Método	Descripción
Insets insets()	Obsoleto. Reemplazado por *getInsets()*.
void invalidate()	Invalida el contenedor.
boolean isAncestorOf(Component c)	Chequea si el componente está contenido en la jerarquía del contenedor.
void layout()	Obsoleto. Reemplazado por *doLayout()*.
void list(PrintStream out, int indent)	Imprime un listado de este contenedor a la salida indicada.
void list(PrintWriter out, int indent)	Visualiza una lista, empezando en la indentación indicada, en la salida indicada.
Component locale(int x, int y)	Obsoleto. Reemplazado por *getComponentAt(int, int)*.
Dimension minimumSize()	Obsoleto. Reemplazado por *getMinimumSize()*.
void paint(Graphics g)	Pinta el contenedor.
void paintComponents(Graphics g)	Pinta cada componente del contenedor.
protected String paramString()	Obtiene la cadena que representa el estado del contenedor.
Dimension preferredSize()	Obsoleto. Reemplazado por *getPreferredSize()*.
void print(Graphics g)	Visualiza el contenedor.
void printComponents(Graphics g)	Visualiza cada uno de los componentes del contenedor.
protected void processContainer Event(ContainerEvent e)	Procesa los eventos del contenedor que ocurren en este, enviándolos a *ContainerListener*.
protected void processEvent (ATEvent e)	Procesa eventos de este contenedor.
void remove(Component comp)	Retira del contenedor el componente indicado.
void remove(int indice)	Retira el componente del contenedor que está en el índice marcado.

Método	Descripción
void removeAll()	Retira todos los componentes de este contenedor.
void removeContainerListener (ContainerListener l)	Retira el *container listener* indicado para que no reciba más eventos *container* de este contenedor.
void removeNotify()	Hace que el contenedor no sea visible retirando su conexión a sus recursos de pantalla nativos.
void setCursor(Cursor cursor)	Fija la imagen del cursor al cursor indicado.
void setFont(Font f)	Fija la fuente del contenedor.
void setLayout(LayoutManager mgr)	Fija el gestor de esquemas para este contenedor.
void update(Graphics g)	Actualiza el contenedor.
void validate()	Valida este contenedor y todos sus subcomponentes.
protected void validateTree()	De forma recursiva, desciende por el árbol del contenedor y recalcula el esquema para todos los subárboles marcados según los necesite (aquellos marcados como inválidos).

Crear *Applets*

"¡Por fin!", dice el programador novato. "Ya estoy preparado para crear una *applet*". "Cierto", le dice. "Ahora, ¿cuál le gustaría?" "Hmm", dice PN.

Usted basa sus *applets* en la clase *java.applet.Applet*, que es en sí misma, una subclase de la clase *java.awt.Container*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Container
          |_____java.awt.Panel
                |____java.applet.Applet
```

Para utilizarlos como referencia, encontrará los métodos de la clase *Applet* en la tabla 6.3.

Tabla 6.3. Métodos de la clase *Applet*.

Método	Descripción
void destroy()	Se le llama cuando nos queremos deshacer de la *applet*.
AppletContext getAppletContext()	Determina el contexto de esta *applet*. El contexto permite a la *applet* solicitar el entorno en el que se ejecuta.
String getAppletInfo()	Obtiene información de esta *applet*
AudioClip getAudioClip(URL url)	Obtiene el objeto *AudioClip* especificado por el argumento URL.
AudioClip getAudioClip(URL url, String nombre*)*	Obtiene el objeto *AudioClip* especificado por los argumentos URL y nombre.
URL getCodeBase()	Obtiene la base URL.
URL getDocumentBase()	Obtiene el documento URL.
Image getImage(URL url)	Obtiene una imagen que puede ser pintada en la pantalla.
Image getImage(URL url, String name)	Obtiene una imagen que puede ser pintada en la pantalla.
Locale getLocale()	Obtiene el *locale* para la *applet*.
String getParameter(String name)	Obtiene el valor del parámetro llamado en la etiqueta HTML.
String[][]getParameterInfo()	Obtiene información de los parámetros de la *applet*.
void init()	Llamado por el *browser* o el visualizador de *applets* para permitir la inicialización de la *applet*
boolean isActive()	Determina si esta *applet* está activa.
static AudioClip newAudioClip (URL url)	Obtiene un *audio clip* de la URL dada.
void play(URL url)	Reproduce el *audio clip* en la URL absoluta especificada.
void play(URL url, String name)	Reproduce el *audio clip* dada la URL y un especificador relativo a él.
void resize(Dimension d)	Solicita que esta *applet* sea redimensionada.

Método	Descripción
void resize(int anchura, *int* altura)	Solicita que esta *applet* sea redimensionada.
void setSub(AppletStub stub)	Establece el *stub* de esta *applet.*
void showStatus(String msg)	Solicita que la cadena sea visualizada en la ventana de estado de la *applet*
void start()	Llamado por el *browser* o visualizador de *applet* para decir a la *applet* que debería empezar la ejecución.
void stop()	Llamado por el *browser* o visualizador de *applet* para decir a la *applet* que debería parar la ejecución.

Veamos un ejemplo. Aquí, se creará la *applet,* que se vio al principio del capítulo, que visualiza el texto "¡Hola desde Java!" en el fichero *applet.java.* Se empieza derivando una nueva clase, *applet*, de la clase *java.applet.Applet*, como sigue:

```
import java.applet.Applet;

public class applet extends Applet
{
    .
    .
    .
}
```

Para visualizar el mensaje en la *applet*, se usará el método *paint*, que la *applet* hereda de la clase *Component*. Cuando se visualiza una *applet*, se llama a su método *paint*, y en ese método, se puede situar el código para dibujar la *applet*. Al método *paint* se le pasa un objeto de la clase *Graphics,* que se verá en los próximos capítulos. Esta es la base del trabajo gráfico en las *applets*. Este objeto soporta un método llamado *drawString*, que se usará para dibujar una cadena de texto en la *applet*. La clase *Graphics* es una clase AWT, por lo que se importarán las clases AWT cuando se sobrescriba el método *Applet paint* por defecto:

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public void paint(Graphics g)
    {
```

```
        .
        .
        .
        }
}
```

Ahora se personaliza el método sobrescrito *paint* para que escriba el texto "¡Hola desde Java!" en la posición (60, 100) en la *applet*. Las coordenadas de la *applet* están en pixels; por lo tanto, (60, 100) está a 60 pixels de la esquina izquierda de la *applet* y 100 pixels más abajo del final de la barra de título. Este es el código:

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}
```

Este nuevo método *paint* dibuja directamente en el área de trabajo de la *applet*. Observe que, también, se puede situar controles, como botones y campos de texto, directamente en esta área de trabajo. También se verá cómo se hace esto en este capítulo. Con *javac*, se puede compilar *applet.java* para obtener *java.class*, y así se acaba de crear una *applet*. Ya es el momento de ver cómo funciona.

Usar la etiqueta HTML <APPLET>

"De acuerdo", dice el programador novato, "he creado una *applet* y la he compilado obteniendo el fichero de extensión "*.class*". ¿Cómo lo veo realmente?" "Hay que usar la etiqueta HTML <APPLET>", le responde.

Una vez que se ha creado un fichero con extensión "*.class*", se puede cargar a un ISP (o verlo en la propia máquina) y fijarle su protección (ver la introducción de este capítulo) para que se pueda leer. A continuación, se puede crear una página Web usando la etiqueta <APPLET> para visualizar la *applet*. Así es el formato de la etiqueta <APPLET>:

```
<APPLET
    [CODEBASE = URL]
    CODE = nombre de fichero
    [ALT = Texto alternativo]
    [NAME = nombre de instancia]
```

```
    WIDTH = pixels
    HEIGHT = pixels
    [ALIGN = alineación]
    [VSPACE = pixels]
    [HSPACE = pixels]
>
[<PARAM NAME = nombre VALUE = valor>]
    .
    .
    .
[<PARAM = NAME nombre VALUE = valor>]
</APPLET>
```

Estos son los atributos de la etiqueta <APPLET>:

- *CODEBASE*: URL que especifica el directorio en el que se busca el código de la *applet*.
- *CODE*: Nombre del fichero de la *applet*, incluyendo la extensión "*.class*".
- *ALT*: Texto que se va a visualizar si un *browser* soporta *applets* pero no puede ejecutarse por alguna razón.
- *NAME*: Nombre de la *applet* en el *browser* Web. Deben darse nombres a las *applets* si se quiere tener la posibilidad de buscar otras *applets* e interactuar con ellas.
- *WIDTH*: La anchura del espacio reservado para la *applet*.
- *HEIGTH*: La altura del espacio reservado para la *applet*.
- *ALIGN:* Especifica la alineación de la *applet*: *LEFT* (izquierda), *RIGHT* (derecha), *TOP* (arriba), *BOTTOM* (abajo), *MIDDLE* (medio), *BASELINE, TEXTOP, ABSMIDDLE* o *ABSBOTTOM*.
- *VSPACE:* Espacio ubicado sobre y desde la *applet*.
- *HSPACE:* Espacio ubicado a la derecha y a la izquierda de la *applet*.
- *PARAM NAME:* Nombre del parámetro que se pasa a la *applet*.
- *PARAM VALUE:* Valor de un parámetro.

Esta es una página Web, *applet.html*, que visualiza la *applet* creada en el apartado anterior (observe que sólo son obligatorios los atributos *CODE*, *WIDTH*, y *HEIGHT*):

```
<HEAD>
 <TITLE>APPLET</TITLE>
 </HEAD>
 <BODY>
 <HR>
```

```
<CENTER>
<APPLET
   CODE=applet.class
   WIDTH=200
   HEIGHT=200
>
</APPLET>
</CENTER>
<HR>
</BODY>
</HTML>
```

En este caso, no se está especificando un código base, por lo tanto se sitúa *applet.class* en el mismo directorio que *applet.html*.

Esta página Web, abierta con *Netscape Navigator,* aparece en la figura 6.1.



Figura 6.1. Una *applet* funcionando con *Netscape Navigator.*

También se puede utilizar el visualizador de *applets* de *Sun*, que viene con Java, para ver la *applet*. Abra la página Web como sigue:

```
C:\>appletviewer applet.html
```

El resultado aparece en la figura 6.2. El visualizador de *applets* siempre soporta la última versión de Java, por si el *browser* Web no lo hace, y no quiere instalar el Java *plug-in.* Siempre se puede usar el visualizador de *applets* para probar sus *applets.*



Figura 6.2. Una *applet* funcionando con el visualizador de *applets*.

Gestionar *browsers* no Java

"Uh-oh", dice el programador novato, "hay un problema. Algunos usuarios están usando un *browser* ligeramente no estándar llamado *SuperWildcat14b...*" "Eso suena ligeramente no estándar", le dice. "Y", continúa el programador novato, "no soporta Java. ¿Hay alguna forma de informar a los usuarios de que están perdiendo algo?" "Claro, "le dice. "Coja una silla y lo veremos".

Si se encierra este texto en el interior de una etiqueta <APPLET>, ese texto será visualizado si el *browser* Web no soporta Java. Por ejemplo, así es como se avisaría a los usuarios de las cosas que se están perdiendo de su *applet*:

```
<APPLET code = "applet.class" width = 100 height = 100>
Lo siento, no tiene Java y por lo tanto, no puede ver mi maravillosa applet.
 </APPLET>
```

Introducir etiquetas <APPLET> en el código

El programador novato suspira. "He estado escribiendo *applets*, y es un poco complicado". "¿Qué quiere decir?" le pregunta. "Escribir una página Web con una etiqueta <APPLET> para probar la salida de la misma, ¿no hay algo más fácil?" "Claro", le dice, "se puede introducir la etiqueta <APPLET> directamente en el código fuente".

Los desarrolladores de *Sun* se dieron cuenta de que algunas veces es molesto tener que crear una página Web para probar una *applet*, por lo que si

se usa el visualizador de *applets*, se puede situar una etiqueta <APPLET> directamente en el fichero de la *applet* con extensión "*.java*" en un comentario, como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
 {
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
 }
```

Después de crear el fichero de extensión "*.class*", se puede iniciar el visualizador de *applets* con el fichero de extensión "*.java*" directamente:

```
C:\>appletviewer applet.html
```

Usar los métodos *init*, *start*, *stop*, *destroy*, *paint* y *update*

El programador novato regresa y dice, "Mi *browser* se ha vuelto graciosillo, ¡dibuja mis *applets* en gris!" "Eso es lo que hacen por defecto muchos *browsers* Web", le dice. "Sin embargo, se puede cambiar añadiendo algún tipo de código de inicialización en la *applet* en el método *init*".

Hay un número importante de métodos de *applets* que se deberían conocer:

- *init*: Es el primer método a llamar; sólo se le llama una vez. Aquí se inicia la *applet*.
- *start:* Llamado después de *init*. A este método se le llama, cada vez que una *applet* aparece de nuevo en la pantalla. Esto es, si el usuario se va a otra página y luego regresa, se llama otra vez al método *start*.

- *stop*: Llamado cuando el *browser* se mueve a otra página. Se puede usar este método para parar la ejecución adicional de *threads* que su *applet* puede haber arrancado.
- *destroy*: Llamado cuando la *applet* va a ser eliminada de la memoria. Se puede ejecutar la limpieza en este momento.
- *paint*: Llamado cuando se va a volver a dibujar la *applet*. Este método pasa un objeto de la clase *Graphics*, y se puede usar en los métodos del objeto para dibujar en la *applet*.
- *update*: Llamado cuando se va a volver a dibujar una parte de la *applet*. La versión por defecto rellena la *applet* con el color de fondo antes de volver a dibujarla, lo que puede llevar a la fluctuación cuando se ejecuta animación, en cuyo caso se sobrescribiría este método.

Se pueden sobrescribir estos métodos para personalizarlos como se quiera. Ya se ha sobrescrito el método *paint* para dibujar una cadena de texto; aquí se sobrescribe el método *init* para cambiar el color de fondo de la *applet* a blanco, usando el método de *applet setBackground* y pasándole el campo *white* de la clase *Color* de Java. Esta *applet*, además, proporciona un esqueleto de una implementación de los otros métodos previamente listados. Este es el código:

```
import java.applet.Applet;
import java.awt.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet
{
    public void init()
    {
        setBackground(Color.white);
    }

    public void start()
    {
    }

    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
```

```
    }

    public void stop()
    {
    }

    public void destroy()
    {
    }
}
```

El método *init* es muy útil, y normalmente se sobrescribe porque permite inicializar su *applet*. En este caso, se ha cambiado el color de fondo de gris (el color por defecto en muchos de los *browsers*) a blanco. En gráficos, hay que destacar a las *applets*. La siguiente sección proporciona una visión general de la gestión de gráficos en las *applets*.

Dibujar gráficos en *applets*

Aprenderá mucho más sobre el dibujo en *applets* en unos pocos capítulos, pero veremos antes algunos de los métodos de gráficos que usaremos aquí:

- *paint*: Llamado cuando se va a volver a dibujar la *applet.*
- *repaint*: Se llama a este método para forzar que la *applet* sea pintada.
- *drawString*: Dibuja una cadena de texto.
- *setBackground*: Fija el color de fondo.
- *setForeground*: Fija el color de primer plano.
- *draw3Drect*: Dibuja un rectángulo 3D.
- *drawBytes*: Dibuja texto, dado un *array* de *bytes*.
- *drawImage*: Dibuja una imagen.
- *drawOval:* Dibuja un óvalo (incluyendo círculos).
- *drawPolyLine:* Dibuja una línea con múltiples segmentos.
- *drawRoundRect:* Dibuja un rectángulo redondeado.
- *drawArc:* Dibuja un arco.
- *drawChars:* Dibuja texto, dado un *array* de caracteres.
- *drawLine*: Dibuja una recta.

- *drawPolygon*: Dibuja un polígono.
- *drawRect*: Dibuja un rectángulo.

Dos métodos que particularmente hay que destacar son el método *paint*, con el que se pinta la *applet*, y el método *repaint*, que fuerza a que se llame al método *paint*.

Usar el *plug-in* de Java del *browser*

"Hey", dice el programador novato, "tengo un problema. Estoy usando los dos grandes *browsers* de Java, pero no soportan las últimas características de Java. ¿Qué puedo hacer?" "Hay una solución fácil", le contesta. "Use el *plug-in* Java".

El *plug-in* Java le permite ejecutar las *applets* de la última versión de Java en *Netscape Navigator* y *Microsot Internet Explorer* implementando el entorno de ejecución de Java como un *plug-in* para *Netscape* y un control *ActiveX* para *Internet Explorer*. Se puede obtener el *plug-in* de Java en *http://java.sun.com/products/plugin*. Se instala automáticamente al instalar JDK.

Para usar las páginas Web con el *plug-in*, se necesita convertir primero su HTML, usando el convertidor HTML de Sun, que se puede obtener en *http://java.sun.com/products/plugin*. El convertidor de HTML es un fichero Java de extensión "*.class*" que se ejecuta en las páginas HTML para convertir la etiqueta <APPLET> de forma que pueda usarse por el *plug-in*. Se puede ver el convertidor de HTML en la figura 6.3.

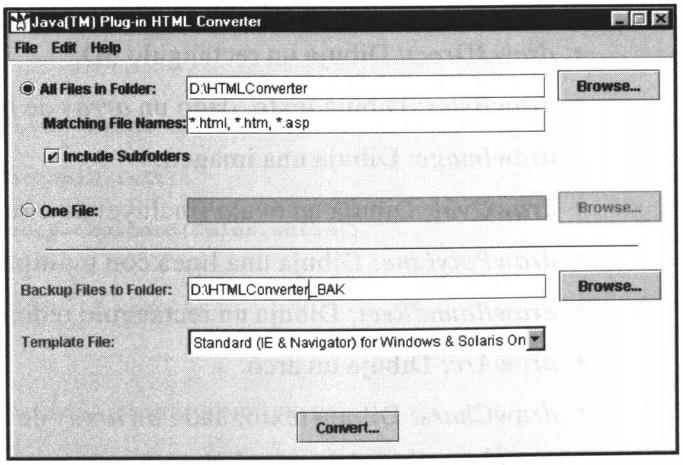


Figura 6.3. El convertidor HTML de Sun.

Para convertir una página Web y usar el *plug-in*, se selecciona un fichero HTML o ficheros en el convertidor HTML y se hace clic sobre el botón ***Convert***. El convertidor cambiará una etiqueta <APPLET> como

```
<APPLET code=adder.class width=200 height=200>>/APPLET>
```

en algo como esto:

```
<!- "CONVERTED_APPLET" ->
<!- CONVERTER VERSION 1.0 ->
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
 WIDTH = 200 HEIGHT = 200     codebase="http://java.sun.com/products/
plugin/1.2/jinstall-12-
win32.cab#Version=1,2,0,0">
<PARAM NAME = CODE VALUE = adder.class >

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2" java_CODE =
adder.class WIDTH = 200 HEIGHT = 200
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html"><NOEMBED></COMMENT>

</NOEMBED></EMBED>
</OBJECT>

<!-
<APPLET  CODE = adder.class WIDTH = 200 HEIGHT = 200 >


</APPLET>
->
<!-"END_CONVERTED_APPLET"->
```

El nuevo fichero HTML usará el *plug-in* Java en lugar de la máquina virtual de Java del *browser* por defecto. Por ejemplo, la figura 6.4 muestra la *applet* de demostración de *Swing TicTacToe* en *http://192.9.48.9/products/plugin/1.2.2/demos/applets/TicTacToe/example1.html* en *Internet Explorer*.

Leer parámetros en *applets*

El gran jefe (GF) aparece mascando un cigarro y dice, "Necesitamos personalizar el saludo de nuestra *applet* para cada cliente". "Pero hay miles de clientes", le dice. "No podemos recompilar la *applet* para cada uno y almacenar cada versión nueva en el sitio Web". "¿Qué sugiere?", pregunta GF.

¿Qué le dirá?

Se puede pasar parámetros a las *applets* en la etiqueta <APPLET>, y el código *applet* puede leer los valores de esos parámetros, lo que significa que

para personalizar la *applet*, sólo necesita dar los diferentes parámetros en la etiqueta <APPLET>. Realmente, para conseguir el valor de un parámetro, se usa el método *getParameter* de la clase *Applet*, pasándole el nombre del parámetro como se especifica en la etiqueta <PARAM>. El método *getParameter* devolverá el valor del parámetro que fue establecido en la etiqueta <PARAM>.

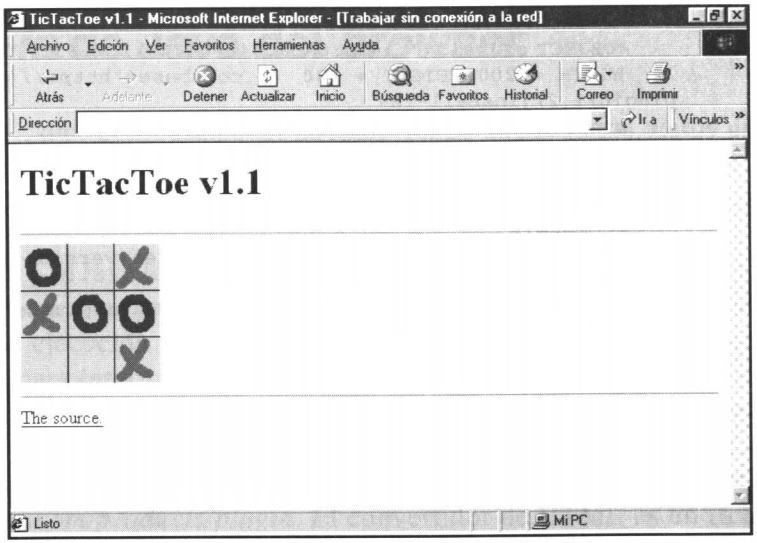


Figura 6.4 Utilizando el *plug-in* de Java.

Este es un ejemplo en el que se pasa un parámetro llamado cadena a una *applet*; el valor de este parámetro es el texto que la *applet* debería visualizar. Así es el código:

```
import java.applet.Applet;
import java.awt.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
    <PARAM NAME = string VALUE = "¡Hola desde Java!">
 </APPLET>
 */


public class applet extends Applet
{
    public void paint(Graphics g)
    {
```

```
            g.drawString(getParameter("cadena"), 60, 100);
        }
    }
```

Usar las consolas de Java en los *browsers*

"Todo este material de *drawString* está bien", dice el programador novato, "pero ¿qué hay de la consola de Java? ¿Y si uso *System.out.println* en una *applet*?" "Eso depende", le dice, "de su *browser*".

Esto es una *applet* que visualiza un mensaje y usa *System.out.println* para visualizar en la consola:

```
import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
        System.out.println("¡Hola desde Java!");
    }
}
```

Si se ejecuta esta *applet* con el visualizador de *applets* Sun, la *applet* se abrirá en una ventana separada, y se verá "¡Hola desde Java!" en la ventana de la consola.

Los *browsers* Web, con frecuencia, tienen también una consola Java, aunque normalmente hay que habilitarlas antes de usarlas. La manera de habilitar la consola Java difiere, desafortunadamente, de un *browser* a otro y de una versión a otra. Actualmente, en *Internet Explorer* se habilita ejecutando el comando Internet del menú Ver, hacer clic sobre la lengüeta Avanzadas, y seleccionar la casilla de activación *Java Console Enabled* (Consola de Java habilitada). La figura 6.5 muestra el resultado de la *applet* anterior tal y como aparece en la consola de Java de *Internet Explorer*, que emerge cuando se visualiza.

En *Netscape Navigator*, se puede abrir la consola de Java ejecutando el comando Java Console (Consola de Java) del menú *Communicator*.

Añadir controles a las *applets*: Campos de texto

"De acuerdo", dice el programador novato, "ahora se puede dibujar texto en una *applet*. Pero, ¿y si quiero que el usuario pueda meter algo de texto?"

"Para eso, " le dice, "se puede usar cualquier clase de controles de texto, como los cuadros de texto".

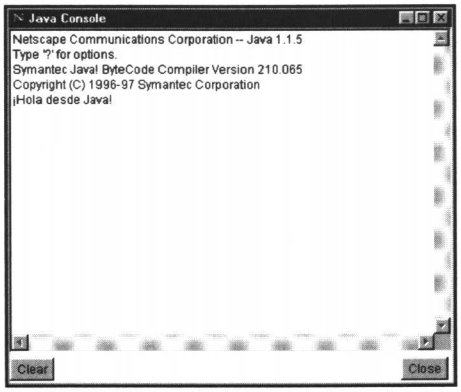


Figura 6.5. Usar la consola de Java de *Internet Explorer*.

Los cuadros de texto están entre los controles más básicos que se pueden usar en AWT. Un cuadro de texto visualiza una línea sencilla y permite al usuario editarlo. Veremos los cuadros de texto, formalmente, en el siguiente capítulo, pero también podremos usarlos aquí al hablar de la gestión de eventos. Esto es un ejemplo de cuadro de texto en el que se crea uno de 20 caracteres en un método *init* de la *applet* (observe que se están importando las clases AWT para poder usar los cuadros de texto):

```
import java.applet.Applet;
import java.awt.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet
{
     public TextField text1;

     public void init()
     {
          text1 = new TextField(20);
          .
          .
          .
     }
}
```

Después de crear un nuevo control, se le debe añadir a la *applet* para que se visualice. He aquí un ejemplo:

```
public void init()
{
    text1 = new TextField(20);
    add(text1);
}
```

El método *add* añade el control al gestor de esquemas actual, que decide donde se debería situar el control (se verán los detalles sobre los gestores de esquemas en el siguiente capítulo). Ahora que el cuadro de texto se ha añadido a la *applet,* se puede situar el texto "¡Hola desde Java!" en el cuadro de texto con el método *setText*, como sigue:

```
public void init()
{
    text1 = new TextField(20);
    add(text1);
    text1.setText("¡Hola desde Java!");
}
```



Figura 6.6. Añadir un campo de texto a una *applet.*

El resultado de este código aparece en la figura 6.6, donde se puede ver el cuadro de texto con el mensaje que se ha puesto en él. El usuario además puede editar este texto. Echaremos un vistazo más profundo a los cuadros de texto en el siguiente capítulo.

Otro control básico es el control de AWT *Button*. Usaremos los botones y los cuadros de texto para discutir la gestión de eventos, por lo tanto, introdu-

ciremos el control *Button* en el siguiente apartado. En el siguiente capítulo, aprenderemos más detalles sobre los botones y los cuadros de texto.

Añadir controles a las *applets*: botones

"Estoy preparado para el siguiente paso", dice el programador novato. "He añadido cuadros de texto a mis *applets*. ¿Qué es lo siguiente?" "Botones", le responde. "Tome una silla y hablaremos de ellos".

Los usuarios pueden hacer clic sobre los botones de su *applet* para indicar que quieren ejecutar alguna acción; por ejemplo, se puede tener un botón llamado "Cambiar color" que, cuando se haga clic sobre él, cambie el color de fondo de la *applet* usando el método *setBackground*. Los botones son soportados por la clase *java.awt.Button*, que discutiremos en detalle en el siguiente capítulo. Es bastante fácil añadir un botón a una *applet*, se hace de la misma forma que se añade un cuadro de texto, como se demostró en el ejemplo anterior. Aquí se está creando y añadiendo un botón con el texto "¡Haga clic aquí!":

```
public class applet extends Applet
{
     TextField text1;
     Button button1;

     public void init()
     {
         text1 = new TextField(20);
         add(text1);
         buton1 = new Button("¡Haga clic aquí!");
         add(button1);
         .
         .
         .
     }
```

El truco, realmente, está en que ocurra algo cuando el usuario haga clic sobre el botón, y para eso, tendremos que echar un vistazo a la gestión de eventos (ver el siguiente punto).

Gestión de eventos

"Hey", dice el programador novato, "he puesto un botón en mi *applet*, pero cuando hago clic sobre él, no ocurre nada. ¿Qué pasa?" "Lo que pasa, " le dice, "es que tiene que implementar la gestión de eventos".

La gestión de eventos, proceso de respuesta que se genera al hacer clic sobre el botón, los movimientos del ratón, etc, ha llegado a ser un tema complejo en Java. Desde Java 1.1, la gestión de eventos ha cambiado significativamente. El modelo actual se llama gestión de eventos delegado. En este modelo, se debe registrar específicamente en Java si se quiere gestionar un evento, como puede ser hacer clic sobre un botón. La idea es que se mejora la ejecución si sólo se informa de los eventos al código que necesita gestionarlos y no al resto.

Los eventos se registran implementando una interfaz de *listener* de eventos. Estos son los eventos de *listeners* disponibles y los tipos de eventos que gestionan:

- *ActionListener:* Gestiona los eventos de acción, como hacer clic sobre los botones.
- *AdjustementListener*: Gestiona los casos en los que un componente es escondido, movido, redimensionado o mostrado.
- *ContainerListener*: Gestiona el caso en el que un componente coge o pierde el foco.
- *ItemListener*: Gestiona el caso en el que cambia el estado de un elemento.
- *KeyListener*: Recibe los eventos de teclado.
- *MouseListener*: Recibe en los casos en que es pulsado el ratón, mete un componente, sale un componente o es presionado.
- *MouseMotionListener*: Recibe en el caso en que se arrastra o mueve el ratón.
- *TextListener*: Recibe los cambios de valor de texto.
- *WindowListener*: Gestiona los casos en que una ventana está activada, desactivada, con o sin forma de icono, abierta, cerrada o se sale de ella.

Cada *listener* es una interfaz, y se deben implementar los métodos de la interfaz (para más detalles sobre las interfaces, ver el capítulo anterior). A cada uno de estos métodos se le pasa un tipo de objeto que corresponde al tipo de evento:

- *ActionEvent*: Gestiona botones, el hacer doble clic en la lista o hacer clic en un elemento del menú.
- *AdjustementEvent*: Gestiona los movimientos de la barra de desplazamiento.

- *ComponentEvent*: Gestiona el caso en el que un componente es escondido, movido, redimensionado o llega a ser visible.
- *FocusEvent*: Gestiona el caso en el que un componente coge o pierde el foco.
- *InputEvent*: Gestiona la marca de activación en una casilla de activación y el hacer clic en un elemento de la lista, hacer selecciones en los controles de opción y las selecciones de los elementos de un menú.
- *KeyEvent*: Gestiona la entrada desde el teclado.
- *MouseEvent*: Gestiona los casos en que se arrastra el ratón, se mueve, se pulsa, se presiona, se suelta o entra o sale un componente.
- *TextEvent*: Gestiona el valor de un cuadro de texto o si ha cambiado.
- *WindowEvent*: Gestiona el caso en el que una ventana está activada, desactivada, en forma de icono, sin forma de icono, abierta, cerrada o abandonada.

Gestión de eventos estándar

Ya es hora de poner a funcionar algo de lo que hemos visto hasta ahora. Empezaremos añadiendo un nuevo botón con el texto "¡Haga clic aquí!" en una *applet,* así como añadir un *listener* de acciones que será notificado cuando se haga clic sobre el botón. Para añadir un *listener* de acciones al botón, se usa el método *addActionListener* del mismo, pasándole un objeto que implementa los métodos de la interfaz *ActionListener*. Este objeto puede ser un objeto de la clase principal de la *applet* o de otra clase. Trataremos ambas variaciones aquí, empezando con el envío de las notificaciones de eventos a la clase *main* de la *applet*.



Truco: también se puede quitar el registro de un *listener* de eventos usando el método *removeListener*.

Aquí vemos cómo se añade un *listener* de acciones a un botón, enviando notificaciones de eventos al objeto de la *applet* actual (observe que indicamos que la clase *applet* ahora implementa la interfaz *ActionListener*):

```
/*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
```

```
 </APPLET>
 */

public class applet extends Applet implements ActionListener
{
     TextField text1;
     Button button1;

     public void init()
     {
         text1 = new TextField(20);
         add(text1);
         button1 = new Button("¡Haga clic aquí!");
         add(button1);
         button1.addActionListener(this);
     }
}
```

Ahora hay que implementar los métodos de la interfaz *ActionListener*. Esta interfaz sólo tiene un método, *actionPerformed*, al que se le pasa un objeto de la clase *ActionEvent* cuando se hace clic sobre el botón:

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

Los objetos *ActionEvents* (que veremos en el siguiente capítulo) heredan un método llamado *getSource* de la clase *EventObject*, y este método devuelve el objeto que produjo el evento. Eso quiere decir que se puede comprobar si este evento fue producido por el botón, *button1*, y en ese caso, situar el texto "¡Hola desde Java!" en el cuadro de texto *text1* de esta forma:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet implements ActionListener
{
     TextField text1;
     Button button1;

     public void init()
     {
         text1 = new TextField(20);
         add(text1);
         button1 = new Button("¡Haga clic aqui!");
```

```
        add(button1);
        button1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event)
    {
        String msg = new String ("¡Hola desde Java!");
        if(event.getSource() == button1){
            text1.setText(msg);
        }
    }
}
```

Esta *applet* aparece en la figura 6.7. Cuando se hace clic sobre el botón, el texto "¡Hola desde Java!" aparece en el cuadro de texto.

La clase que se registra para recibir los eventos no necesita ser la clase principal de la *applet* (y de hecho, los desarrolladores de Sun originalmente intentaron que no lo fuera, aunque ahora es la práctica común). Veremos el uso de otras clases para recibir eventos, próximamente.

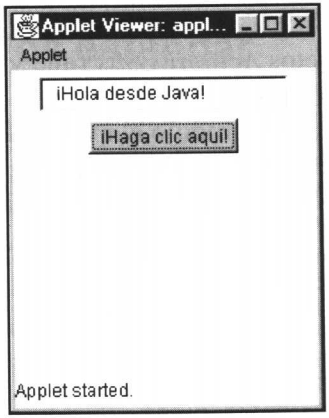


Figura 6.7. Soporte de hacer clic sobre botones.

Uso de clases delegadas

Veamos un ejemplo en el que se está creando una nueva clase para implementar la interfaz *ActionListener*.

Observe que esto es un poco pesado, porque se quiere trabajar con el cuadro de texto y los controles de botón en el objeto principal de la *applet*, por lo que hay que pasar y almacenar referencias a ese objeto en el nuevo constructor de la clase:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet
{
     public TextField text1;
     public Button button1;

     public void init()
     {
         text1 = new TextField(20);
         add(text1);
         button1 = new Button("¡Haga clic aqui!");
         add(button1);
         a obj = new a(this);
         button1.addActionListener(obj);
     }
}

class a implements ActionListener {
   applet c;

   a(applet appletobject)
   {
        c = appletobject;
   }

   public void actionPerformed(ActionEvent event)
   {
        String msg = new String ("¡Hola desde Java!");
        if(event.getSource() == c.button1){
           c.text1.setText(msg);
        }
    }
}
```

Este código funciona igual que la versión anterior de esta *applet,* salvo que internamente usa una nueva clase para gestionar eventos, no la clase principal de la *applet*. Hay veces que esto es útil, como cuando se tienen muchos eventos para gestionar y no se quiere agrandar la clase principal de la *applet*.

Además, hay otras formas de determinar el objeto que causó el evento, por ejemplo, se pueden usar comandos.

Uso de los comandos de acción

Java permite asociar comandos con eventos causados por los botones AWT y los elementos del menú. Cuando se trabaja con botones, el comando por defecto es la inscripción del botón (veremos cómo se crean los comandos personalizados en el siguiente capítulo), así podemos determinar en qué botón se ha hecho clic mirando su inscripción (no es una buena idea si su programa cambia las inscripciones). Se puede obtener el comando para un botón con el método *getActionCommand*. Aquí vemos cómo se implementa la *applet* anterior usando los comandos:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet implements ActionListener {

     TextField text1;
     Button button1;

     public void init()
     {
         text1 = new TextField(20);
         add(text1);
         button1 = new Button("¡Haga clic aquí!");
         add(button1);
         button1.addActionListener(this);
     }

    public void actionPerformed(ActionEvent event)
    {
        String msg = new String ("¡Hola desde Java!");
        String caption = event.getActionCommand();

        if(caption.equals("¡Haga clic aquí!")){
            text1.setText(msg);
        }
    }
}
```

Esto le introduce en el proceso de la gestión de eventos de una manera moderna. Sin embargo, la antigua forma de Java 1.0 todavía se ejecuta en Java, aunque se considera obsoleto. Por motivos deexhaustividad, le echaremos un vistazo en el siguiente punto.

La forma antigua de gestionar eventos

Java 1.0 usa un acercamiento no delegado a los eventos. En el modelo de eventos de Java 1.0, no se necesita registrar la obtención de eventos, se pasan de cualquier forma, y se pueden gestionar en un método llamado *action*. Por ejemplo, aquí tenemos cómo aparecería la *applet* anterior usando el modelo de eventos antiguo:

```
import java.applet.Applet;
import java.awt.*;

public class First extends Applet
{
    TextField text1;
    Button button1;

    public void init()
    {
        text1 = new TextField(20);
        add(text1);
        button1 = new Button("Púlseme");
        add(button1);
    }

    public boolean action (Event e, Object o)
    {
        String caption = (String)o;
        String msg = "¡Hola desde Java!";
        if(e.target instanceof Button){
            if(caption == "Púlseme"){
                text1.setText(msg);
            }
        }
        return true;
    }
}
```

El problema era que el método *action* con frecuencia llegaba a ser enorme, por lo que los diseñadores de Java introdujeron el modelo de eventos delegado en el que se pueden pasar eventos donde se quiera si estar restringidos a un método *action*.

De hecho, hay otra forma de gestionar eventos, por extensión de los componentes.

Extender componentes

Si le gusta ser furtivo, se pueden gestionar eventos derivando nuevas clases de componentes y sobrescribir los métodos de los componentes que

gestionan eventos. De hecho, esta forma de hacer las cosas termina con la técnica de gestión de eventos de Java 1.0, porque toda la gestión de eventos tiene lugar en un método; sin embargo, esto es descorazonador. A pesar de ello, lo veremos aquí para completar el tema.

Aquí vemos cómo implementamos la *applet* extendiendo la clase *Button* en una clase derivada llamada *newButton*. En esta nueva clase, sobrescribiremos el método *processActionEvent* de la clase *Button*, visualizando el texto "¡Hola desde Java!" en el campo de texto de la *applet* después de llamar al método *processActionEvent* de la clase *Button*:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

class newButton extends Button
{
     applet a;

     newButton(applet aref, String s)
     {
          super(s);
         a = aref;
        enableEvents(AWTEvent.ACTION_EVENT_MASK);
     }
     protected void processActionEvent (ActionEvent e)
     {
           super.processActionEvent(e);
           a.text1.setText("¡Hola desde Java!");
     }
}
```

Ahora todo lo que tenemos que hacer es crear un nuevo botón de esta clase y añadirlo a la *applet*:

```
public class applet extends Applet {

     TextField text1;
     Button button1;

     public void init()
     {
         text1 = new TextField(20);
         add(text1);
```

```
            button1 = new newButton(this, "¡Haga clic aquí!");
            add(button1);
        }
}
```

Eso es todo; ahora esta *applet* funciona como las otras.

Usar las clases adaptador

"Ugh", dice el programador novato, "entiendo que el modelo de eventos delegado lo utilizan las interfaces *listener*, pero algunas veces es una pena tener que implementar todos los métodos de una interfaz cuando se quiere usar uno solo". "Es cierto", le dice, "por eso, Sun introdujo las clases adaptador, para hacer que todo el proceso sea mucho más fácil".

Las clases adaptador son clases que ya han sido implementadas en varias de las interfaces de eventos disponibles. Cada método de una interfaz está implementado como método vacío, sin ningún código, en una clase adaptador, y todo lo que se necesita hacer es sobrescribir el método o métodos que se quiera.

Veamos un ejemplo. En este caso, empezaremos con una *applet* que almacena una cadena, "¡Hola desde Java!", por defecto, y visualiza esa cadena en el método *paint*. Luego se va a añadir un *listener* de ratón (aprenderá más sobre ello en el siguiente capítulo) a la *applet,* para que cuando el usuario haga clic en la *applet*, aparezca el mensaje "¡Hola a Java!". Al igual que la interfaz *ActionListener*, la interfaz *MouseListener* tiene cinco métodos que deben implementarse. Sin embargo, se quiere usar el método *mouseClicked* para gestionar el hecho de que se haga clic en el ratón, por lo que usaremos la clase *MouseAdapter* en su lugar.

Empezaremos añadiendo una clase que implementa un *listener* de ratón al programa. Esta clase crea una subclase de la clase *MouseAdapter*, y lo llamaremos *ma*:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet
```

```
{
    public String s = "¡Hola desde Java!";

    public void init() {
        addMouseListener(new ma(this));
    }

    public void paint(Graphics g)
    {
        g.drawString(s, 60, 100);
    }
}
```

Se pasa un objeto de la clase principal de la *applet* al constructor de la clase *ma* por lo que se puede llegar a los campos de la *applet*. Cuando se hace clic en el ratón, se reemplaza la cadena de texto en la *applet* con el texto "¡Hola a Java!" y luego se repinta la *applet,* haciendo que la nueva cadena aparezca en la pantalla:

```
class ma extends MouseAdapter {
    applet a;

    ma(applet appletobject)
    {
        a = appletobject;
    }

    public void mouseClicked(MouseEvent me)
    {
         a.s = "¡Hola a Java!";
         a.repaint();
    }
}
```

El resultado de este código aparece en la figura 6.8, observe que esto era un poco complicado porque teníamos que pasar el objeto *applet* al constructor de la clase ma para que la clase pueda alcanzar los campos de la clase *applet.*



Figura 6.8. Hacer clic sobre el botón con las clases adaptador.

Por otro lado, las clases internas tienen acceso a los campos de sus clases encerradas, por lo que los adaptadores están, con frecuencia, implementados como clases internas. Veremos esto a continuación.

Usar clases adaptador internas anónimas

En el capítulo anterior vimos las clases internas anónimas. Estas clases tienen el siguiente formato especial:

```
new SuperType(constructor parameters)
{
    //métodos y datos
}
```

Aquí, *SuperType* es la clase o interfaz de la clase interna anónima que es derivada. En este caso, se va a derivar una clase interna anónima de la clase *MouseAdapter* para implementar la *applet* que vimos en el apartado anterior. Observe que se está sobrescribiendo el método *mouseClicked* como se hizo en el punto anterior, pero ya es hora de usar una clase interna anónima, que hace que el código sea mucho más compacto:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

public class applet extends Applet
{
    public String s = "¡Hola desde Java!";

    public void init() {
        addMouseListener(new MouseAdapter(){
        public void mousePressed(MouseEvent me) {
          s = "¡Hola a Java!";
          repaint();
        }});
    }

    public void paint(Graphics g)
    {
        g.drawString(s, 60, 100);
```

```
        }
    }
```

Crear ventanas de aplicación

El programador novato aparece y dice, "De acuerdo, ahora se pueden crear *applets*, pero ¿qué hay de las aplicaciones que usan ventanas?" "Correcto, " le dice.

Al igual que se escribían *applets*, cuando se escribe una aplicación de ventana, se es responsable de crear su propia ventana en la que se visualiza esa aplicación. El tipo de ventana más común para usar esto es la ventana de Java *Frame*, soportada en la clase *java.awt.Frame*. Este es el diagrama de herencia para esa clase (esta clase se deriva de la clase *java.awt.Window*, que trataremos dentro de unos pocos capítulos):

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Container
          |____java.awt.Window
               |____java.awt.Frame
```

Encontrará los constructores de la clase *Frame* en la tabla 6.4 y sus métodos en la tabla 6.5.



Truco: una cosa importante que observar: aunque el gestor del esquema por defecto de la clase *Applet* es el gestor del flujo, en la clase *Frame* es el borde. Si se quiere usar otro esquema cuando se añaden componentes a una ventana *Frame*, tendremos que cambiar el gestor de esquemas en sí mismo.

He aquí un ejemplo en el que se deriva una clase nueva, *AppFrame*, de la clase *Frame* y se personaliza para visualizar la cadena "¡Hola desde Java!", para hacer que aparezca la *applet* desarrollada anteriormente.

Tabla 6.4. Constructores de la clase *Frame*.

Constructor	Descripción
Frame()	Construye una nueva instancia de *Frame* que es inicialmente invisible.
Frame(String título)	Construye un nuevo objeto *Frame* que es invisible, con el título dado.

Tabla 6.5. Métodos de la clase Frame.

Método	Descripción
void addNotify()	Hace que este marco sea visible conectándolo a un recurso nativo de pantalla.
protected void finalize()	Llamado cuando el *frame* va a ser eliminado por la colección *garbage*.
int getCursorType()	Obsoleto. Reemplazado por *Component.getCursor()*.
static Frame[]getFrames()	Obtiene un *array* que contiene todos los *frames* creados por la aplicación.
Image getIconImage()	Obtiene la imagen que se va a visualizar en el icono minimizado.
MenuBar getMenuBar()	Obtiene la barra de menú.
int getState()	Obtiene el estado de la ventana.
String getTitle()	Obtiene el título de la ventana.
boolean isResizable()	Indica si esta ventana es redimensionable por el usuario.
protected String paramString()	Devuelve el parámetro *String* de esta ventana.
void remove(MenuComponent m)	Retira la barra de menú especificada de esta ventana.
void removeNotify()	Hace que este *frame* no sea visible, retirando su conexión de sus recursos de pantalla nativos.
void setCursor(int cursorType)	Obsoleto. Reemplazado por *Component.setCursor(Cursor)*.
void setIconImage(Image image)	Fija la imagen que se va a visualizar en el icono minimizado de esta ventana.
void setMenuBar(MenuBar mb)	Fija la barra de menú de esta ventana para que sea la barra de menú indicada.
void setResizable(boolean resizable)	Establece si esta ventana es redimensionable por el usuario.
void setState(int estado)	Fija el estado de esta ventana.
void setTitle(String título)	Fija el título de esta ventana a la cadena indicada.

Como con la clase *java.applet.Applet,* la clase *java.awt.Frame* es derivada de la clase *java.awt.Component*, se puede usar el método *paint* para visualizar gráficos en la clase *Frame*. De hecho, el método *paint* sería como lo hemos creado anteriormente en el fichero de la aplicación, *app.java*. Este es el código:

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

class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}
```

Se necesita un método *main* para arrancar la aplicación, por lo que creamos ese método en una nueva clase llamada *app*:

```
public class app
{
    public static void main(String [] args)
    {
            .
            .
            .
    }
}
```

Para visualizar el *frame* de una ventana de aplicación, se crea un objeto de la clase *AppFrame,* como sigue:

```
public class app
{
    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();
         .
         .
         .
    }
}
```

Ahora se da al objeto un tamaño de 200 x 200 pixels y se visualiza en la pantalla con el método *show:*

```
public class app
{
    public static void main(String [] args)
    {
```

```
        AppFrame f = new AppFrame();

        f.setSize(200, 200);

        f.show();
    }
}
```



Truco: si no se da al *frame* de una ventana un tamaño antes de visualizarlo, sólo se verá una barra de título; en términos de interfaz gráfica de usuario (GUI), no habrá área de trabajo, que es el área de debajo de la barra de título en el que se visualiza la aplicación.



Figura 6.9. Crear una ventana de aplicación.

Los resultados de este código aparecen en la figura 6.9. Como se puede ver, el mensaje "¡Hola desde Java!" aparece en la aplicación, como se intentaba. Además se pueden añadir controles a esta aplicación, como se hizo con la *applet* del botón antes, en este capítulo.

Hay algo más que mencionar aquí: primero, si se lanza una aplicación con la utilidad java (es decir, como *java app*), esa herramienta esperará hasta que la aplicación devuelva el control a la consola. Desde el punto de vista del usuario, la consola parece estar colgada hasta que se abandone la aplicación. Si se quiere controlar el retorno inmediatamente a la consola después de que la aplicación sea lanzada y visualice su propia ventana, hay que usar la utilidad *javaw* en su lugar. Ver el capítulo 1 para más información.

Otro punto importante es que si se quieren distribuir las aplicaciones a los usuarios que no tienen el JDK instalado, se puede usar el entorno de ejecución de Java (JRE). De nuevo, ver el capítulo 1 para más detalles.

Finalmente, es importante apuntar que no hay forma fácil de salir de la aplicación en la figura 6.9; hacer clic sobre el botón Cerrar no tiene efecto. Se puede pulsar **Ctrl-C** para finalizar la ejecución de la herramienta java en la ventana de la consola, pero eso es bastante complicado. En su lugar, se tiene

que gestionar los eventos de cierre de ventana para terminar la aplicación cuando la ventana se cierra. Echemos un vistazo a eso en el siguiente punto.

Salir de una aplicación al cerrar su ventana

"Hey", dice el programador novato, "fue un poco difícil acabar con mi primera ventana de aplicación, hacer clic sobre el botón Cerrar no tiene ningún efecto". "Entonces, ¿cómo se termina la aplicación?", le pregunta. "Apagando mi ordenador", dice el PN suspirando.

Java espera que se gestione el caso en el que el usuario haga clic en el botón **Cerrar** en una ventana de aplicación (aunque esto hubiera sido más fácil con ventanas *Swing*). Para finalizar una aplicación cuando el usuario haga clic en el botón **Cerrar**, se deben capturar los eventos ventana, con la ayuda de la interfaz *WindowListener*. Aquí veremos una forma compacta de hacerlo, modificando la aplicación desarrollada en el capítulo anterior usando la clase *WindowAdapter*, que implementa la interfaz *WindowListener* con una implementación vacía de cada método. En este caso, usaremos una clase adaptador interna anónima (ver el apartado correspondiente en este capítulo para más detalles) y sobrescribir el evento *windowClosing*. En ese método del evento, añadiremos código para salir de la aplicación *System.exit(0)*. Esto finaliza la aplicación con un código de salida de 0 (que significa una terminación normal). Este es el código:

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

class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}

public class app
{
    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new WindowAdapter() {public void
          windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
```

```
    }
}
```

Esto es todo. Ahora cuando se hace clic en el botón **Cerrar** de la aplicación, la ventana se cierra y la aplicación finaliza.

Aplicaciones que se pueden ejecutar como *applets*

El gran jefe (GF) llega y dice, "tenemos que recortar los costes de desarrollo. De ahora en adelante, todas las *applets* deben ser también aplicaciones". "Hmm", le dice, "¿para cuándo tiene que estar eso?" "¿No está todavía? pregunta GF.

Si se añade un método *main* a una *pplet,* esa *applet* se puede ejecutar como *applet* y aplicación; Java ignorará el método *main* cuando se ejecuta como una *applet*, y se ejecutará el método *main* cuando se ejecuta como una aplicación.

Este es un ejemplo en el que se han combinado la *applet* y la aplicación desarrollada en este capítulo en un programa:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

 /*
 <APPLET
    CODE=applet.class
    WIDTH=200
    HEIGHT=200 >
 </APPLET>
 */

import java.applet.Applet;
import java.awt.*;

public class applet extends Applet
{
    public static void main(String [] args)
    {
        AppFrame f = new AppFrame();

        f.setSize(200, 200);

        f.addWindowListener(new WindowAdapter() {public void
            windowClosing(WindowEvent e) {System.exit(0);}});

        f.show();
```

```
    }

    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}

class AppFrame extends Frame
{
    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", 60, 100);
    }
}
```

Este código se puede ejecutar como *applet* y también como aplicación (el gran jefe estará orgulloso).

7 AWT: Cuadros de texto, botones, casillas de activación y plantillas

Este capítulo trata un número importante de componentes AWT: cuadros de texto, botones, casillas de activación y botones de opción. Ahora que ya estamos habituados a los componentes visibles, echaremos un vistazo a los gestores de esquemas o plantillas de Java que permiten ordenar los componentes en una *applet* o aplicación. Además revisaremos los paneles en Java, que permiten unir componentes en una superficie y mandarla al gestor de esquemas. Empezaremos echando un vistazo a los cuadros de texto.

Cuadros de texto

Los cuadros de texto son los componentes básicos de AWT para soportar texto. Estos componentes gestionan cadenas de texto de una dimensión; permiten visualizar el texto que el usuario escribe, poner máscaras de texto cuando se introduce una clave secreta, leer el texto que el usuario ha incluido y mucho más. Dentro de AWT, estos componentes son los fundamentales, junto con los botones.

Botones

Los botones proporcionan al usuario una forma rápida de iniciar alguna acción. Hay que hacer clic sobre todos ellos. Cualquier usuario está familiarizado con los botones y ya vimos cómo funcionan al discutir la gestión de eventos en el capítulo anterior. A los botones se les puede dar una inscripción, como "¡Púlseme!". Cuando el usuario hace clic sobre el botón, el código recibe una notificación, siempre que se haya registrado la gestión de eventos desde el botón.

Casillas de activación

Las casillas de activación son como los botones, salvo que tienen un doble estado, es decir, pueden estar seleccionadas o no. Cuando se seleccionan, aparecen con algún tipo de marca, como puede ser una marca de activación o una X (el tipo de indicación visual depende del sistema operativo en que se haya programado AWT, que es una de las razones por las que *Sun* introdujo *Swing*, que puede visualizar componentes con la misma forma independientemente del sistema operativo). El usuario puede marcar una casilla de activación para seleccionar un tipo de opción, como los ingredientes de un *sandwich*, habilitar la revisión de ortografía automáticamente o habilitar la impresión mientras se está haciendo otra cosa. Se usan las casillas de activación para permitir al usuario opciones de selección no exclusivas; por ejemplo, la revisión automática de ortografía y la impresión en *background* pueden estar habilitados al mismo tiempo. Los botones de opción, sin embargo, son otra historia.

Botones de opción

Utilizando los botones de opción, se puede permitir al usuario seleccionar una entre un conjundo de opciones mutuamente excluyentes. Sólo una de esas opciones puede seleccionarse al mismo tiempo. Por ejemplo, usando botones de opción, se puede dejar al usuario seleccionar un color de impresión y el día de la semana. En AWT, los botones de opción son un tipo de casilla de activación, y cuando se seleccionan, visualizan un punto redondo, un cuadrado o algún otro tipo de indicación (de nuevo, la indicación visual depende del sistema operativo). Veremos cómo funcionan en este capítulo.

Hemos añadido componentes a las *applets* y aplicaciones usando el método *add*. Este método es, realmente, un método del gestor de esquemas que hay por defecto, el *flow layout manager*. Por defecto, este gestor es el responsable de ordenar los componentes en las *applets* AWT. El *flow layout manager* ordena los componentes de la misma forma que un procesador de texto debería ordenar las palabras a lo largo de la página y pasar a la siguiente línea cuando sea necesario, creando lo que *Sun* llama un flujo de componentes. Veremos que se pueden personalizar los esquemas de flujo de forma extensiva. Sin embargo, las limitaciones son claras, especialmente si se trata de mantener alguna posición de componentes respecto a otros, porque si el usuario cambia el tamaño de la *applet* o aplicación, todos los componentes tendrán que moverse. Por otro lado, hay otros gestores de esquemas en AWT (y algunos nuevos en *Swing*), y veremos el *grid* AWT, *border, card* y *grid layout* en este capítulo.

¿Por qué no se puede poner un componente donde se quiera y luego olvidarse de él? Los programadores novatos de Java con frecuencia se frustran al tratar con los gestores de esquemas de AWT, y quieren saber por qué no pueden dar las coordenadas de los componentes que quieren utilizar. De hecho, se puede, aunque cada uno sería responsable de gestionar el caso en el que las ventanas sean redimensionadas y hacer que los componentes se muevan. Para posicionar los componentes donde se quiera, se puede indicar que no se quiere ningún gestor de esquemas, y luego medir y ubicar los componentes como se quiera, usando *add* para visualizarlos como sigue:

```
setLayout(null);
text1 = new TextField(20);
text1.setSize(200, 50);
text1.setLocation(20, 20);
add(text1);
```

Esto añade un cuadro de texto de tamaño (200, 50) en la localización (20, 20) de un contenedor, como por ejemplo una ventana de una *applet*. Como veremos, se pueden añadir componentes a contenedores sin un gestor de esquemas, algo que es útil tener en mente si los esquemas de AWT le frustran demasiado.

Un contenedor AWT muy útil es el componente *Panel*. Se pueden ordenar los componentes en un panel y luego añadir el panel, en sí mismo, al esquema de una *applet* o aplicación. En este capítulo veremos cómo se hace.

Esto es todo. Ahora que ya hemos revisado lo que veremos en este capítulo, es el momento de pasar a la siguiente sección.

Usar cuadros de texto

"Hey", dice el programador novato, "quiero permitir a los usuarios escribir una clave secreta, pero ese maldito Johnson se queda mirando por encima del hombro de la gente y ve lo que escriben". "Eso tiene fácil arreglo", le dice. "Basta con poner el carácter de *echo* del cuadro de texto a un asterisco u otro carácter similar. ¡Problema resuelto!"

En el capítulo anterior, vimos y usamos los cuadros de texto; estos componentes pueden visualizar una sola línea de texto, y el usuario puede editarlo. Este es el diagrama de herencia de la clase de cuadros de texto, *TextField*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.TextComponent
          |____java.awt.TextField
```

Se pueden ver los constructores de esta clase en la tabla 7.1 y sus métodos en la 7.2.

Tabla 7.1. Constructores de la clase *TextField.*

Constructor	Descripción
TextField()	Construye un nuevo cuadro de texto.
TextField(int columnas)	Construye un nuevo cuadro de texto vacío con el número de columnas indicado.
TextField(String texto)	Construye un nuevo cuadro de texto con el texto indicado.
TextField(String texto, *int* columnas)	Construye un nuevo cuadro de texto inicializado con el texto indicado y con el número de columnas especificado.

Tabla 7.2. Métodos de la clase *TextField.*

Método	Descripción
void addActionListener (ActionListener l)	Añade el *ActionListener* indicado para recibir eventos.
void addNotify()	Crea el compañero del objeto cuadro de texto.
boolean echoCharIsSet()	Indica si el cuadro de texto tiene puesto un carácter para utilizarlo como *echo.*

Método	Descripción
int getColumns()	Obtiene el número de columnas del cuadro de texto.
char getEchoChar()	Obtiene el carácter que se utiliza como *echo.*
Dimension getMinimumSize()	Obtiene las dimensiones mínimas para el cuadro de texto.
Dimension getMinimumSize (int columnas*)*	Obtiene las dimensiones mínimas de un cuadro de texto con el número de columnas indicado.
Dimension getPreferredSize()	Obtiene el tamaño preferido del cuadro de texto.
Dimension getPreferredSize (int columnas*)*	Obtiene el tamaño preferido del cuadro de texto con el número de columnas indicado.
Dimension minimumSize()	Obsoleto. Reemplazado por *getMinimum-Size().*
Dimension minimumSize (int columnas*)*	Obsoleto. Reemplazado por *getMinimum-Size(int).*
protected String paramString()	Obtiene la representación en cadena del estado del cuadro de texto.
Dimension preferredSize()	Obsoleto. Reemplazado por *getPreferred-Size().*
Dimension preferredSize (int columnas*)*	Obsoleto. Reemplazado por *getPreferred-Size(int).*
protected void processAction-Event(ActionEvent e)	Procesa los eventos ocurridos en el cuadro de texto, enviándolos a los objetos *Action-Listener.*
protected void processEvent (AWTEvent e)	Procesa eventos en el cuadro de texto.
void removeActionListener (ActionListener l)	Elimina el *action listener* indicado, para que no reciba más eventos.
void setColumns(int columnas*)*	Establece el número de columnas del cuadro de texto.
void setEchoChar(char c)	Establece el carácter *echo* para el cuadro de texto.

Método	Descripción
void setEchoCharacter(char c)	Obsoleto. Reemplazado por *setEchoChar-(char).*
void setText(String t)	Indica el texto que va en el cuadro de texto.

Veamos un ejemplo. En este caso, se creará un cuadro de texto para una clave que visualizará un asterisco (*) cada vez que el usuario escriba un carácter. Puede que se pregunte cómo se podrá leer la clave escrita. La respuesta es utilizando el método *getText* del cuadro de texto, que hereda de la clase *Component*. De hecho, se añadirá un segundo cuadro de texto en este programa y se visualizará la clave cuando el usuario pulse la tecla **Intro**. Empezamos añadiendo dos cuadros de texto, de 30 caracteres cada uno:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=clave.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class clave extends Applet implements ActionListener
{
     public TextField text1;
     public TextField text2;

     public void init()
     {
          text1 = new TextField(30);
          add(text1);
          text2 = new TextField(30);
          add(text2);
          .
          .
          .
```

A continuación, se establece que el carácter *echo* en *text1* (el cuadro de texto de la clave) es * y se añade un *action listener* a ese cuadro de texto:

```
public void init()
     {
          text1 = new TextField(30);
          add(text1);
          text2 = new TextField(30);
```

```
        add(text2);

        text1.setEchoChar('*');
        text1.addActionListener(this);
    }
```

Cuando el usuario pulsa la tecla **Intro**, se llama al método *actionPerformed*, por lo que sobrescribimos dicho método para establecer que el texto en el segundo cuadro de texto es el del componente de la clave:

```
public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == text1){
            text2.setText(text1.getText());
        }
    }
```

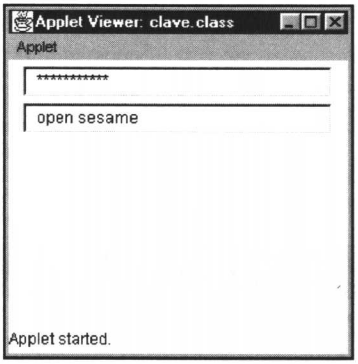


Figura 7.1. Leer claves en cuadros de texto.

El resultado de este código se muestra en la figura 7.1. Cuando el usuario escribe una clave en el cuadro de texto de la parte superior y pulsa la tecla **Intro**, esa clave aparece en el otro cuadro de texto (no es exactamente lo que se llamaría seguridad). Se puede ver este ejemplo en el CD de ejemplo de este libro, en el fichero clave.java.



Truco: otro método de cuadro de texto que también es útil es *getSelectedText*, que permite obtener el texto seleccionado del cuadro de texto.

Usar etiquetas

Las etiquetas AWT son como los cuadros de texto, salvo que el usuario no puede editarlas. Se pueden utilizar las etiquetas para presentar texto que no se

puede editar, o, como su nombre indica, para etiquetar otros componentes. Este es el diagrama de herencia de la clase *Label*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Label
```

En la tabla 7.3 se pueden ver los constructores de la clase *Label,* y sus métodos en la tabla 7.4.

Tabla 7.3. Constructores de la clase *Label.*

Constructor	Descripción
Label()	Construye una etiqueta vacía.
Label(String texto)	Construye una nueva etiqueta con el texto indicado, justificado a la izquierda.
Label(String texto, *int* alineación)	Construye una nueva etiqueta que presenta la cadena especificada con la alineación indicada.

Tabla 7.4. Métodos de la clase *Label*

Método	Descripción
void addNotify()	Crea el compañero de esta etiqueta.
int getAlignment()	Obtiene la alineación actual de esta etiqueta.
String getText()	Obtiene el texto de esta etiqueta.
protected String paramString()	Devuelve la cadena que representa el estado de la etiqueta.
void setAlignment(int alineación)	Fija la alineación de esta etiqueta a la que se especifica.
void setText(String text)	Establece el texto de esta etiqueta al texto indicado.

El texto de una etiqueta se puede justificar pasándole al constructor de la etiqueta los campos *Label.LEFT, Label.CENTER y Label.RIGHT.*

Esto es un ejemplo que crea tres etiquetas con las posibles alineaciones de texto:

```
import java.applet.Applet;
import java.awt.*;
```

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

/*
<APPLET
    CODE=etiqueta.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class etiqueta extends Applet
{
     Label label1;
     Label label2;
     Label label3;

     public void init()
     {
          label1 = new Label("¡Hola desde Java!", Label. LEFT);
          add(label1);
          label2 = new Label("¡Hola desde Java!", Label. CENTER);
          add(label2);
          label3 = new Label("¡Hola desde Java!", Label. RIGHT);
          add(label3);
     }
}
```

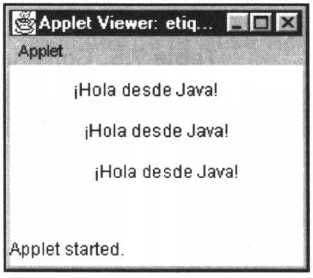


Figura 7.2. Justificación de texto en una etiqueta.

El resultado de esta *applet* se muestra en la figura 7.2. Este ejemplo se puede encontrar en el CD, en el fichero etiqueta.java.

Usar botones

"Quiero hacer que los usuarios interactúen con mi programa", dice el programador novato. " Quiero dejarles que indiquen lo que quieren hacer simplemente con un clic de ratón, quiero que puedan seleccionar una acción rápida y fácilmente, quiero..." "Botones", le contesta. "Lo que quiere son botones". "Cierto, " responde PN.

Todo usuario de GUI está familiarizado con los botones, esos controles elementales sobre los que se hace clic para indicar a un programa que debe empezar a realizar alguna acción; por ejemplo, podría permitir al usuario hacer clic sobre un botón para cambiar el color de fondo de una aplicación. Los botones están soportados en la clase *java.awt.Button*. Esta es la jerarquía de la clase:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Button
```

Los constructores de la clase *Button* se muestran en la tabla 7.5, y sus métodos en la tabla 7.6.

Para gestionar los eventos de los botones se usa la interfaz *ActionListener*, como vimos en el capítulo anterior. Esta interfaz tiene un único método, *actionPerformed*, al que se le pasa un objeto de la clase *ActionEvent* cuando se hace clic sobre el botón:

```
void actionPerformed(ActionEvent e)
{
     .
     .
     .
}
```

Tabla 7.5. Constructores de la clase *Button*.

Constructor	Descripción
Button	Construye un botón sin etiqueta.
Button(String *etiqueta)*	Construye un botón con la etiqueta indicada.

Tabla 7.6. Métodos de la clase *Button*.

Método	Descripción
void addActionListener(ActionListener l)	Añade el *ActionListener* indicado para recibir eventos del botón.
void addNotify()	Crea el compañero del botón.
String getActionCommand()	Obtiene el comando del evento producido por el botón.
String getLabel()	Obtiene la etiqueta del botón.
protected String paramString()	Obtiene la cadena que representa el estado del botón

Método	Descripción
protected void processActionEvent-(ActionEvent e)	Procesa los eventos que tienen lugar en el botón, enviándolos a los objetos *ActionListener* registrados.
protected void processEvent (AWTEvent e)	Procesa eventos del botón.
void removeActionListener(Action-Listener l)	Elimina el *actionListener* para que no pueda recibir eventos del botón.
void setActionCommand(String comando*)*	Establece el nombre del comando para el evento producido por el botón.
void setLabel(String etiqueta)	Fija la etiqueta del botón para ser la cadena indicada.

Este es el diagrama de la herencia de la clase *ActionEvent*:

```
java.lang.Object
|____java.util.EventObject
     |____java.awt.AWTEvent
          |____java.awt.event.ActionEvent
```

Todos los métodos de la clase *ActionEvent* se muestran en la tabla 7.7.

Tabla 7.7. Métodos de la clase *ActionEvent*.

Método	Descripción
String getActionCommand()	Obtiene la cadena del comando.
int getModifiers()	Obtiene las claves del modificador, mantenidas durante el evento.
String paramString()	Obtiene una cadena que identifica el evento.

Como vimos en el capítulo anterior, hay dos formas principales de determinar qué botón se seleccionó, usando el método *getSource* y usando comandos. Primero, veremos cómo se hace esto con *getSource*.He aquí un ejemplo con un botón que, cuando se hace clic sobre él, se visualiza el mensaje "¡Hola desde Java!" en un cuadro de texto (observe que se ha registrado un *action listener* con el botón y se investiga cuál es el botón sobre el que se ha hecho clic usando el método *getSource* antes de poner el texto apropiado en el cuadro de texto).

```
import java.applet.Applet;
import java.awt.*;
```

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

public class boton extends Applet implements ActionListener {

/*
<APPLET
    CODE=boton.class
    WIDTH=200
    HEIGHT=200 >

*/
<APPLET
    TextField text1;
    Button button1;

    public void init()
    {
        text1 = new TextField(20);
        add(text1);
        button1 = new Button("¡Haga clic aqui!");
        add(button1);
        buton1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event)
    {
        String msg = new String ("¡Hola desde Java!");
        if(event.getSource() == button1){
            text1.setText(msg);
         }
    }
}
```

El resultado de este código se muestra en la figura 7.3. Esta *applet* se puede encontrar en el CD, boton.*java*.



Figura 7.3. Gestionar el hacer clic en los botones.

También se pueden usar comandos con botones; en el capítulo anterior, vimos que el comando asociado por defecto a cada botón es su inscripción. Sin embargo, dado que estas inscripciones pueden cambiar durante la ejecución del programa (por ejemplo, usando el método *setLabel)*, se debería asociar un comando con el botón en sí mismo. Se puede dar a un botón un comando de tipo *string* con el método *setActionCommand* como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=boton2.class
    WIDTH=200
    HEIGHT=200 >

*/
<APPLET
public class boton2 extends Applet implements ActionListener {

    TextField text1;
    Button button1;

    public void init()
    {
        text1 = new TextField(20);
        add(text1);
        button1 = new Button("¡Haga clic aquí!");
        add(button1);
        button1.setActionCommand("Botón apretado");
        button1.addActionListener(this);
    }
    .
    .
    .
}
```

Ahora, en el método *actionPerformed,* se puede obtener el comando para el botón sobre el que se hizo clic, usando el método *getActionCommand.* Si el botón sobre el que se hizo clic es el de la derecha, se puede situar el mensaje en el cuadro de texto, como sigue:

```
public void actionPerformed(ActionEvent event)
    {
        String msg = new String ("¡Hola desde Java!");
        String command = event.getActionCommand();

        if(command.equals("Botón apretado")){
            text1.setText(msg);
        }
```

```
    }
}
```

Esta *applet* se puede encontrar en el CD en boton2.*java*.

Usar casillas de activación

"Ahora, tengo otro problema", dice el programador novato. "Quiero que los usuarios puedan seleccionar de qué quieren una pizza, por lo que me gustaría que cuando se haga clic sobre un botón, éste aparezca pulsado para que los usuarios sepan lo que ya han seleccionado". "No hay ningún problema", le dice. "No utilice botones". "¿No?" pregunta PN. "No, " le dice. "Utilice casillas de activación en su lugar".

Una casilla de activación permite al usuario seleccionar opciones. Cuando el usuario hace clic en una casilla de activación, algún tipo de indicación visual, como una marca de activación (el indicador varía con el sistema operativo cuando se usa AWT), se utiliza para indicar que la opción está seleccionada. Si se hace clic otra vez en la casilla de activación, se deselecciona. En AWT, las casillas de activación están soportadas por la clase *java.awt.Checkbox,* que tiene el siguiente diagrama de herencia:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Checkbox
```

Tabla 7.8. Constructores de la clase *Checkbox*.

Constructor	Descripción
Checkbox()	Crea una casilla de activación sin etiquetas.
Checkbox(String etiqueta)	Crea una casilla de activación con la etiqueta especificada.
Checkbox(String etiqueta, *boolean* estado)	Crea una casilla de activación con la etiqueta especificada y fija el estado.
Checkbox(String etiqueta, *boolean* estado, *CheckboxGroup* grupo)	Crea una casilla de activación en el grupo indicado y fija el estado.
Checkbox(String etiqueta, *CheckboxGroup* grupo, *boolean* estado)	Construye una casilla de activación con la etiqueta indicada en el grupo especificado.

Tabla 7.9. Métodos de la clase *CheckBox*.

Método	Descripción
void addItemListener(ItemListener l)	Añade el *item listener* indicado a la casilla de activación.
void addNotify()	Crea el compañero de la casilla de activación.
CheckboxGroup getCheckbox-Group()	Crea un grupo de casillas de activación.
String getLabel()	Obtiene la etiqueta de la casilla de activación.
Object[]getSelectedObjects()	Obtiene un *array* (de longitud 1) que contiene la etiqueta de la casilla de activación (o *null* si no se ha seleccionado).
boolean getState()	Determina si la casilla de activación está en estado seleccionado o no seleccionado.
protected String paramString()	Obtiene una cadena que representa el estado de la casilla de activación.
protected void processEvent (AWTEvent e)	Procesa eventos en la casilla de activación.
protected void processItemEvent (ItemEvent e)	Procesa los eventos de *items* que se producen en la casilla de activación, enviándolos a cualquiera de los objetos *ItemListener* registrados.
void removeItemListener (ItemListener l)	Retira el *item listener* indicado, para que no reciba más eventos de la casilla de activación.
void setCheckboxGroup (CheckboxGroup g)	Establece el grupo de casillas de activación para el grupo dado.
void setLabel(String etiqueta)	Fija la etiqueta de la casilla de activación en esta cadena.
void setState(boolean state)	Fija el estado de la casilla de activación en el estado dado.

En la tabla 7.8 se pueden encontrar los constructores de la clase *Checkbox* y sus métodos en la tabla 7.9. Observe, en particular, que se puede poner el

estado de una casilla de activación con *setState* y obtener el estado con *getState*.

Veamos un ejemplo en el que se añaden cuatro casillas de activación a una *applet*, y cuando el usuario hace clic sobre una de ellas, se utiliza un cuadro de texto para indicar que se ha seleccionado. Observe que las casillas de activación no utilizan la interfaz *ActionListener* como lo hacen los botones; en su lugar, usan la interfaz *ItemListener*, que sirve para gestionar los componentes que pueden ser seleccionados o deseleccionados. La interfaz *ItemListener* sólo tiene un método, *itemStateChanged*, al que se le pasa un parámetro de la clase *ItemEvent*:

```
void itemStateChanged(ItemEvent e)
```

Los métodos de la clase *ItemEvent* se pueden encontrar en la tabla 7.10.

Así es como se añaden casillas de activación en una *applet* y un *ItemListener* a cada una de ellas:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=casillas.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class casillas extends Applet implements ItemListener {

    Checkbox checkbox1, checkbox2, checkbox3, checkbox4;
    TextField text1;

    public void init()
    {
        checkbox1 = new Checkbox("1");
        add(checkbox1);
        checkbox1.addItemListener(this);
        checkbox2 = new Checkbox("2");
```

Tabla 7.10. Métodos de la clase *ItemEvent*.

Método	Descripción
Object getItem()	Obtiene el elemento afectado por el evento.
ItemSelectable getItemSelectable()	Obtiene el originador del evento.

Método	Descripción
int getStateChange()	Obtiene el tipo de cambio de estado (es decir, seleccionado o quitada la selección).
String paramString()	Obtiene la cadena que coincide con el evento.

```
add(checkbox2):
        checkbox2.addItemListener(this);
        checkbox3 = new Checkbox("3");
        add(checkbox3):
        checkbox3.addItemListener(this);
        checkbox4 = new Checkbox("4");
        add(checkbox4):
        checkbox4.addItemListener(this);
        text1 = new TextField(30);
        add(text1);
    }
```

Ahora, sobrescribimos el método *itemStateChanged*, para determinar sobre qué casilla de activación se pulsó, usando el método *getItemSelectable* del objeto *ItemEvent*:

```
public void itemStateChanged(ItemEvent e)
    {
        if(e.getItemSelectable() == checkbox1){
            text1.setText("¡Casilla de activación 1 pulsada!");
        } else if(e.getItemSelectable() == checkbox2){
            text1.setText("¡Casilla de activación 2 pulsada!");
        } else if(e.getItemSelectable() == checkbox3){
            text1.setText("¡Casilla de activación 3 pulsada!");
        } else if(e.getItemSelectable() == checkbox4){
            text1.setText("¡Casilla de activación 4 pulsada!");
    }
}
```

Ahora, ya sabemos cómo utilizar el método *getStateChanged* del objeto *ItemEvent* para determinar si una casilla de activación está seleccionada o no. Este método devuelve *Checkbox.SELECTED* o *Checkbox.DESELECTED*. Y, por supuesto, se puede usar el método *getState* de la casilla de activación para hacer la misma determinación. Además se puede establecer el estado de la casilla de activación con el método *setState*.

El resultado de esta *applet* se muestra en la figura 7.4. Se puede encontrar en el CD, en casillas.*java*.

Debido a que es tedioso tener tantas sentencias *if* en la escala *if-else* del código anterior, se puede visualizar únicamente la casilla de activación sobre la que se ha hecho clic obteniendo su etiqueta directamente, como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
```



Figura 7.4. Gestionar el hacer clic sobre las casillas de activación.

```
/*
<APPLET
    CODE=casillas2.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class casillas2 extends Applet implements ItemListener {

    Checkbox checkbox1, checkbox2, checkbox3, checkbox4;
    TextField text1;

    public void init()
    {
        checkbox1 = new Checkbox("1");
        add(checkbox1);
        checkbox1.addItemListener(this);
        checkbox2 = new Checkbox("2");
        add(checkbox2);
        checkbox2.addItemListener(this);
        checkbox3 = new Checkbox("3");
        add(checkbox3);
        checkbox3.addItemListener(this);
        checkbox4 = new Checkbox("4");
        add(checkbox4);
        checkbox4.addItemListener(this);
        text1 = new TextField(35);
        add(text1);
    }

    public void itemStateChanged(ItemEvent e)
    {
        text1.setText("¡Casilla de activación " +
```

```
                ((Checkbox) e.getItemSelectable()).getLabel() + "pulsa-
da!");
        }
    }
```

Esta nueva versión de la *applet* aparece en casillas.*java* en el CD. Hay otro tipo de casilla de activación que se puede utilizar, los botones de opción, y los revisaremos en la siguiente sección.

Usar botones de opción

"Uh-oh", dice el programador novato, "tengo otro problema. Puse casillas de activación en mi programa para que los usuarios pudieran seleccionar el día de la semana, y un usuario seleccionó miércoles y viernes". "Bien", le contesta, "debería usar botones de opción, en lugar de casillas de activación para visualizar opciones excluyentes, como el día de la semana".

En la programación AWT, los botones de opción son un tipo especial de casilla de activación, y se usan en grupos. Sólo un botón de opción de un grupo puede ser seleccionado de una vez; cuando el usuario selecciona un botón de opción en un grupo, el resto del grupo es automáticamente desactivado.

Cuando se añaden casillas de activación a un grupo, se convierten en botones de opción de forma automática. AWT soporta grupos de casillas de activación con la clase *CheckboxGroup*. Esta clase sólo tiene un constructor, *CheckboxGroup*, que no tiene parámetros, cuyos métodos se verán en la tabla 7.11. Observe que, puesto que los botones de opción son realmente casillas de activación, se pueden usar los métodos *Checkbox*, como *getState* y *setState*.

Tabla 7.11. Métodos de la clase *CheckboxGroup*.

Método	Descripción
Checkbox getCurrent()	Obsoleto. Sustituido por *getSelectedCheckBox()*.
Checkbox getSelectedCheckbox()	Obtiene la casilla de activación que actualmente esta seleccionada dentro del grupo.
void setCurrent(Checkbox box)	Obsoleto. Reemplazado por *setSelectedCheckbox(Checkbox)*.
void setSelectedCheckbox (Checkbox box)	Establece la casilla de activación actualmente seleccionada en este grupo.

Método	Descripción
String toString()	Devuelve una representación tipo *string* del grupo, incluyendo el valor de su selección actual.

Observe, por ejemplo, que se puede determinar qué botón de opción está seleccionado en un grupo, con el método *getSelectedCheckbox* de la clase *CheckboxGroup*, y se puede inicializar con uno seleccionado con el método *setSelectedCheckbox*.

Este es un ejemplo en el que se crea un grupo de casillas de activación llamado *radios,* y se añaden cuatro botones de opción a ese grupo. Se añade un botón de opción a un grupo de casillas de activación añadiéndolo al grupo, pasándolo como parámetro al constructor de la casilla de activación, que convierte la casilla de activación en un botón de opción. Este es el código:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=botondeopcion.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class botondeopcion extends Applet implements ItemListener {

    CheckboxGroup radios;
    Checkbox radio1, radio2, radio3, radio4;
    TextField text1;

    public void init()
    {
        radios = new CheckboxGroup();

        radio1 = new CheckboxGroup("1", false, radio);
        add(radio1);
        radio1.addItemListener(this);

        radio2 = new CheckboxGroup("2", false, radio);
        add(radio2);
        radio2.addItemListener(this);

        radio3 = new CheckboxGroup("3", false, radio);
        add(radio3);
        radio3.addItemListener(this);

        radio4 = new CheckboxGroup("4", false, radio);
```

```
        add(radio4);
        radio4.addItemListener(this);

        text1 = new TextField(25);
        add8text1);
    }
```

Observe que se ha añadido un *ItemListener* a cada botón de opción, por lo que se puede implementar el método *itemStateChanged* para indicar qué botón de opción ha sido seleccionado, como sigue:

```
public void itemStateChanged(ItemEvent e)
    {
        text1.setText("¡Botón de opción " +
              ((Checkbox) e.getItemSelectable()).getLabel() + "pulsa-
do!");
    }
}
```

El resultado de este código se muestra en la figura 7.5, y podrá encontrar esta *applet* en el CD como botondeopcion.java.

Ahora que estamos trabajando con componentes en programas visuales, es hora de tener en cuenta cómo se ordenan estos componentes, y trataremos de hacerlo en los siguientes puntos.



Figura 7.5. Gestionar el hacer clic en botones de opción.

Esquemas de flujo *(flow layout)*

"Uh-oh", dice el programador novato, "de nuevo Java está graciosillo". "¿Qué ha hecho ahora?" le pregunta. PN dice, "Bien, estoy creando una calculadora para multiplicar y quiero que todos los cuadros de texto se almacenen verticalmente, pero se ordenan en filas". "Eso se debe a que está utilizando un esquema de flujo *(flow layout)*", le dice.

Java tiene un número de gestores de esquemas AWT que gestionan cómo se visualizan y se instalan los componentes en los contenedores. Por defecto, las *applets* usan el *flow manager*, que ordena los componentes igual que lo hacen los procesadores con el texto, incluyendo la ruptura de la palabra al pasar de línea.

Tabla 7.12. Constructores de la clase *FlowLayout*.

Constructor	Descripción
FlowLayout()	Construye un nuevo *flow layout* con alineación centrada. El margen horizontal y vertical será de 5 pixels.
FlowLayout(int align)	Construye un nuevo *flow layout* con la alineación dada. El margen horizontal y vertical será de 5 pixels.
FlowLayout(int align, int hgap, int vhap)	Crea un nuevo *flow layout*, con la alineación dada y los márgenes horizontal y vertical dados entre los componentes.

Tabla 7.13. Métodos de la clase *FlowLayout*.

Método	Descripción
void addLayoutComponent(String nombre, *Component comp)*	Añade al esquema el componente. especificado.
int getAlignment()	Obtiene la alineación del esquema.
int getHgap()	Obtiene la separación horizontal entre los componentes.
int getVhap()	Obtiene la separación vertical entre los componentes.
void layoutContainer(Container target)	Pone el contenedor.
Dimension minimumLayoutSize (Container target)	Devuelve las dimensiones mínimas necesarias para poner los componentes en el contenedor *target.*
Dimension preferredLayoutSize (Container target)	Devuelve las dimensiones preferidas para el esquema, dados los componentes en el contenedor *target.*

Método	Descripción
void removeLayoutComponent (Component comp)	Elimina el componente del esquema.
void setAlignment(int align)	Fija la alineación.
void setHgap(int hgap)	Fija la separación horizontal entre los componentes.
void setVgap(int vgap)	Fija la separación vertical entre los componentes.
String toString()	Devuelve la representación de tipo *string* de este objeto *FlowLayout*.

Los constructores de la clase *FlowLayout* aparecen en la tabla 7.12 y sus métodos en la tabla 7.13.

Para tener algún control sobre cómo el *flow layout* ordena los componentes, se puede especificar su alineación utilizando los siguientes campos *FlowLayout*:

- *CENTER*: Indica que cada fila de componentes debería estar centrada.
- *LEADING:* Indica que cada fila de componentes debería estar justificada a la primera esquina del contenedor.
- *LEFT:* Indica que cada fila de componentes debería estar justificada a la izquierda.
- *RIGHT:* Indica que cada fila de componentes debería estar justificada a la derecha.
- *TRAILING:* Indica que cada fila de componentes debería estar justificada a la siguiente esquina del contenedor.

Por defecto, el *flow layout* centra cada componente de la fila, pero aquí podemos ver cómo se crea un nuevo *flow layout* que justifica los componentes a la derecha usando el método *setLayout*:

```
import java.applet.Applet;
import java.awt.*;

/*
  <APPLET
     CODE=flow.class
     WIDTH=200
     HEIGHT=200 >
   </APPLET>
```

```
*/

public class flow extends Applet {

    TextField text1, text2, text3;

    public void init(){
        setLayout(new FlowLayout(FlowLayout.RIGHT));
        text1 = new TextField(10);
        add(text1);
        text2 = new TextField(10);
        add(text2);
        text3 = new TextField(10);
        add(text3);
    }
}
```

Una vez que se han añadido algunos cuadros de texto a este nuevo gestor, se puede ver que están justificados a la derecha, como se muestra en la figura 7.6.

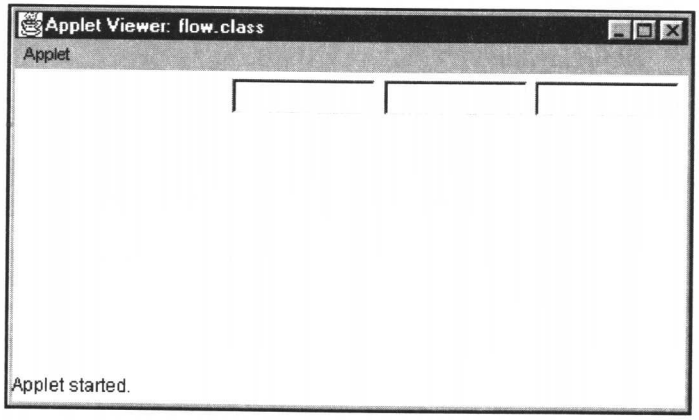


Figura 7.6. Componentes de un *flow layout* justificados a la derecha.

Sin embargo, algunas veces un *flow layout* no es correcto. Por ejemplo, echemos un vistazo a esta *applet*, que presenta la calculadora que el programador novato estaba intentando crear:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
  <APPLET
     CODE=multiplicadora.class
     WIDTH=200
     HEIGHT=200 >
   </APPLET>
```

```
*/
public class multiplicadora extends Applet implements ActionListener {

    TextField text1, text2, text3;
    Label multiplylabel;
    Button b1;

    public void init()
    {
        text1 = new TextField(10);
        add(text1);

        multiplylabel = new Label("*");
        add(multiplylabel);

        text2 = new TextField(10);
        add(text2);

        b1 = new Button("=");
        add(b1);
        b1.addActionListener(this);

        text3 = new TextField(10);
        add(text3);
    }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource() == b1){
            int product = Integer.parseInt(text1.getText()) *
                Integer.parseInt(text2.getText());
            text3.setText(String.valueOf(product));
        }
    }
}
```

El resultado se muestra en la figura 7.7, y esta *applet* está en multiplicadora.java en el CD. Como se puede ver en la figura, los cuadros de texto están ordenados según el *flow layout*, y el resultado es lo que el programador novato buscaba. Echaremos un vistazo a otro gestor de esquemas, el *grid layout*, próximamente.

Grid layouts

"Entonces, ¿cómo arreglo mi calculadora para multiplicar? Los cuadros de texto no están en su sitio", pregunta el programador novato. "Tiene que usar un gestor de esquemas diferente, " le contesta, "el *grid layout*".

El *grid layout* permite añadir componentes a un contenedor, posicionándolos en una cuadrícula. Al *grid layout* se le dan las dimensiones de la cuadrícula, como por ejemplo, cinco filas y cinco columnas, y cuando se

añadan componentes a este esquema, se hará empezando por la izquierda de la primera fila.

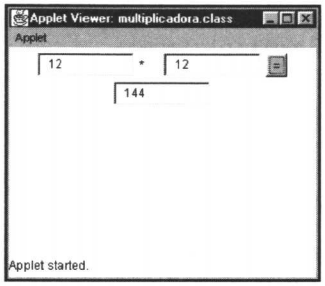


Figura 7.7. Primer intento de una calculadora para multiplicar.

Una vez que la primera fila esté rellena, los componentes se sitúan en la primera columna de la segunda fila y así sucesivamente. Observe que cada componente tiene el mismo tamaño y dimensiones. Este es el diagrama de herencia para el *grid layout*:

```
java.lang.Object
|____java.awt.GridLayout
```

En la tabla 7.14 se encuentran los constructores de la clase *GridLayout* y sus métodos en la tabla 7.15.

Tabla 7.14. Constructores de la clase *GridLayout.*

Constructor	Descripción
GridLayout()	Crea un *grid layout* en una fila, con una columna por componente.
GridLayout(int filas, *int* columnas)	Crea un *grid layout* con el número de filas y de columnas indicados.
GridLayout(int filas, *int* columnas, *int hgap, int vgap)*	Crea un *grid layout* con el número de filas, columnas y separación dados.

Tabla 7.15. Métodos de la clase *GridLayout.*

Método	Descripción
void addLayoutComponent(String nombre, *Component* comp)dado	Añade al esquema el componente.
int getColumns()	Obtiene el número de columnas del esquema.

Método	Descripción
int getHgap()	Obtiene la separación horizontal entre los componentes.
int getVgap()	Obtiene la separación vertical entre los componentes.
void layoutContainer(Container padre)	Pone el contenedor usando el esquema.
*Dimension minimumLayoutSize (Container*padre)	Determina el tamaño mínimo del contenedor usando el *grid layout.*
*Dimension preferredLayoutSize (Container*padre)	Determina el tamaño preferido del contenedor usando el *grid layout.*
void removeLayoutComponent (Component comp)	Elimina del esquema el componente indicado.
void setColumns(int cols)	Establece el número de columnas en el esquema.
void setHgap(int hgap)	Establece la separación horizontal entre los componentes.
void setVgap(int vgap)	Establece la separación vertical entre los componentes.
void setRows(int filas)	Establece el número de filas del esquema.
String toString()	Obtiene una representación de tipo *string* del esquema.

Este es un ejemplo en el que se crea una cuadrícula de cinco filas y una columna, pasando esas dimensiones al constructor del gestor *grid layout* como *GridLayout(5,1)*. Cuando se inicia la adición de componentes a este esquema, serán almacenados uno encima de otro.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=multiplicadora2.class
    WIDTH=200
    HEIGHT=200 >

*/
```

```
</APPLETa
public class multiplicadora2 extends Applet implements ActionListener {

    TextField text1, text2, text3;
    Label multiplylabel;
    Button button1;

    public void init()
    {
        setLayout(new GridLayout(5,1));

        text1 = new TextField(10);
        add(text1);

        multiplylabel = new Label("*", Label.CENTER);
        add(multiplylabel);

        text2 = new TextField(10);
        add(text2);

        button1 = new Button("=");
        add(button1);
        button1.addActionListener(this);

        text3 = new TextField(10);
        add(text3);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == button1){
            int product = Integer.parseInt(text1.getText()) *
                Integer.parseInt(text2.getText());
            text3.setText(String.valueOf(product));
        }
    }
}
```

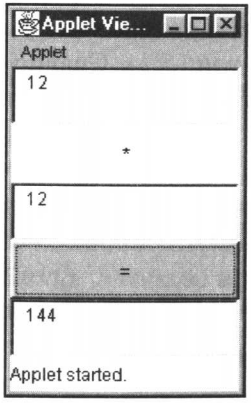


Figura 7.8. Segunda versión de la calculadora para multiplicar.

El resultado de este código se muestra en la figura 7.8. Como se puede ver en la figura, los componentes de esta *applet* se almacenan verticalmente, como se suponía.

Una forma de trabajar con esquemas es dividir los componentes del programa en paneles y añadir dichos paneles a los esquemas. Esto hace que se tenga más control sobre dónde van a ir los controles. Echaremos un vistazo a los paneles en el siguiente punto.

Usar paneles

El programador novato todavía no está satisfecho con los problemas que tiene con los esquemas y dice, "Necesito afinar el control en mi esquema, tengo muchos componentes que visualizar". "De acuerdo, " le dice, "para tener más control, puede usar los paneles de Java". "¡Estupendo! Cuénteme todo eso", contesta PN.

La clase *Panel* de Java es un contenedor que se puede usar en los esquemas. Se añaden componentes a un panel y luego se añade ese panel al esquema.

Dado que los paneles son contenedores, se pueden fijar sus gestores usando los métodos *setLayout*. Los paneles, por sí solos, no son contenedores visuales, no tienen bordes, por ejemplo, y sólo existen para contener componentes. Este es el diagrama de herencia para la clase *Panel*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Container
          |____java.awt.Panel
```

El constructor de la clase *Panel* se muestra en la tabla 7.16 y su método en la tabla 7.17.

Tabla 7.16. Constructor de la clase *Panel*.

Constructor	Descripción
Panel(LayoutManager layout)	Crea un nuevo panel con el *layout* especificado.

Tabla 7.17. Método de la clase *Panel*.

Método	Descripción
addNotify()	Crea el compañero del panel.

Este es un ejemplo en el que se deriva una nueva clase, *buttonPanel*, de la clase *Panel*, y se añaden algunos botones al panel en el constructor *buttonpanel*:

```
import java.applet.Applet;
import java.awt.*;

/*
<APPLET
    CODE=paneles.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

class buttonpanel extends Panel
{
    Button button1, button2, button3, button4;

    buttonpanel()
    {
        button1 = new Button("1");
        add(button1);
        button2 = new Button("2");
        add(button2);
        button3 = new Button("3");
        add(button3);
        button4 = new Button("4");
        add(button4);
    }
}
```

Ahora, se pueden ordenar paneles de esta nueva clase en un *grid layout*, como sigue:

```
public class paneles extends Applet {

    buttonpanel panel1, panel2, panel3, panel4, panel5, panel6;

    public void init(){
        setLayout(new GridLayout(2,3));

       panel1 = new buttonpanel();
       panel2 = new buttonpanel();
       panel3 = new buttonpanel();
       panel4 = new buttonpanel();
       panel5 = new buttonpanel();
       panel6 = new buttonpanel();

       add(panel1);
       add(panel2);
       add(panel3);
       add(panel4);
       add(panel5);
       add(panel6);
    }
}
```

El resultado de este código se muestra en la figura 7.9, y el código de esta *applet* se puede encontrar en paneles.java en el CD. Como se puede ver en esta figura, se ha añadido cada panel al *grid layout*; usando paneles se da un control más fino al esquema.

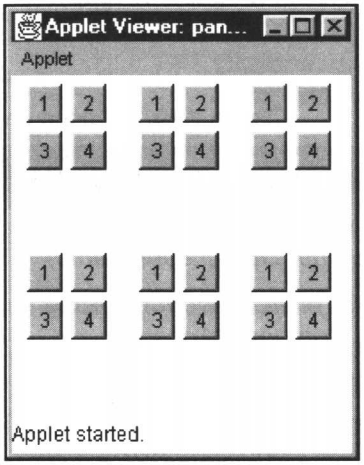


Figura 7.9. Ordenar componentes usando paneles.

Border Layout

El programador novato vuelve y dice, "¿hay más esquemas que deba conocer?" "Por supuesto, " le dice. "Debería conocer el *border layout*, por ejemplo". PN pregunta, "¿Ordena los límites?" "Un poco", le dice.

El *border layout* permite ordenar los componentes alrededor del borde de un contenedor. Es útil si se quieren soportar, digamos, barras de desplazamiento y desplazar un componente central. Los *border layouts* son las ventanas de AWT por defecto, ventanas *frame*, y cuadros de diálogo. Se indica dónde se quiere que vaya un componente pasando al constructor *BorderLayout* cadenas como "*North*", "*East*", etc. Este es el diagrama de herencia para *BorderLayout*:

```
java.lang.Object
|____java.awt.BorderLayout
```

Los constructores de la clase *BorderLayout* se encuentran en la tabla 7.18 y sus métodos en la tabla 7.19.

Tabla 7.18. Constructores de la clase *BorderLayout*.

Constructor	Descripción
BorderLayout()	Construye un nuevo *border layout*.
BorderLayout(int hgap, int vgap)	Construye un *border layout* con las separaciones entre componentes indicadas.

Tabla 7.19. Métodos de la clase *BorderLayout*.

Método	Descripción
void addLayoutComponent(Component comp, Object constraints)	Añade el componente al *layout*, usando el objeto dado.
void addLayoutComponent(String nombre, *Component comp)*	Añade al *layout* el componente dado, con el nombre indicado.
int getHgap()	Obtiene la separación horizontal entre componentes.
float getLayoutAlignmentX(Container padre)	Obtiene la alineación a lo largo del eje x.
float getLayoutAlignmentY(Container padre)	Obtiene la alineación a lo largo del eje y.
int getVgap()	Obtiene la separación vertical entre componentes.
void invalidateLayout(Container target)	Invalida el esquema.
void layoutContainer(Container target)	Pone el contenedor.
Dimension maximumLayoutSize (Container target)	Obtiene las dimensiones máximas del esquema dados los componentes en el *target*.
Dimension minimumLayoutSize (Container target)	Obtiene las dimensiones mínimas del esquema usando el gestor de esquemas.
Dimension preferredLayoutSize (Container target)	Obtiene las dimensiones preferidas del esquema usando el gestor de esquemas.
void removeLayoutComponent (Component comp)	Elimina del esquema el componente dado.

Método	Descripción
void setHgap(int hgap)	Establece la separación horizontal entre los componentes.
void setVgap(int vgap)	Establece la separación vertical entre los componentes.
String toString()	Obtiene una representación tipo *string* del esquema.

Este es un ejemplo en el que se crea un panel que contiene un cuadro de texto y se añaden cuatro botones alrededor del borde del panel. Cuando el usuario hace clic sobre un botón, la *applet* indicará, en el cuadro de texto, el botón que se pulsó.

Así es como se crea el panel que visualiza un cuadro de texto:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
  <APPLET
    CODE=border.class
    WIDTH=200
    HEIGHT=200 >
  </APPLET>
*/

class textPanel extends Panel
{
     TextField Text1;

    textPanel(){
        Text1 = new TextField(30);
        add(Text1);
    }
}
```

Ahora se crea un *border layout,* situando el panel del cuadro de texto en el centro y añadiéndole cuatro botones alrededor del borde. Se puede especificar dónde se quiere situar un componente con cadenas como "*North*", "*West*", "*Center*", etc. Se crea el esquema como sigue:

```
public class border extends Applet implements ActionListener
{

    Button button1, button2, button3, button4;
    textPanel Panel1;

    public void init()
```

```
    {
        setLayout(new BorderLayout());

        button1 = new Button("1");
        add("North", button1);
        button1.addActionListener(this);

        button2 = new Button("2");
        add("West", button2);
        button2.addActionListener(this);

        button3 = new Button("3");
        add("South", button3);
        button3.addActionListener(this);

        button4 = new Button("4");
        add("East", button4);
        button4.addActionListener(this);

        Panel1 = new textPanel();
        add("Center", Panel1);
        Panel1.Text1.setLocation(0, 0);
    }

    public void actionPerformed(ActionEvent e)
    {
        Panel1.Text1.setText("Botón " +
            ((Button) e.getSource()).getLabel() + " pulsado.");
    }
}
```

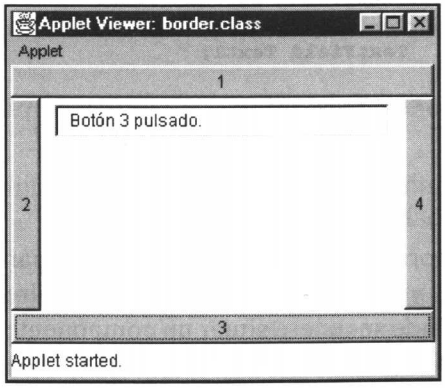


Figura 7.10. Crear un *border layout.*

El resultado de este código se muestra en la figura 7.10. Como se puede ver en esta figura, los botones aparecen alrededor del perímetro central del panel.

Card Layouts

"¿Hay más esquemas AWT?" dice el programador novato. "Claro que los hay", le dice, "*card layout*". "Veámoslo", dice PN.

El gestor *card layout* visualiza los contenedores que se le pasan como tarjetas. A cada una se le da un nombre; luego se puede mover de una a la otra con el método *show* de *card layout*. Este es el diagrama de herencia para la clase *CardLayout*:

```
java.lang.Object
|____java.awt.CardLayout
```

Además del método *show*, se pueden visualizar tarjetas específicas usando los métodos *first, last, next* y *previous* de la clase *CardLayout*. Los constructores de la clase *CardLayout* se encuentran en la tabla 7.20 y sus métodos en la tabla 7.21.

Tabla 7.20. Constructores de la clase *CardLayout*.

Constructor	Descripción
CardLayout()	Crea un nuevo *card layout*.
CardLyout(int hgap, int vgap)	Crea un nuevo *card layout* con las separaciones horizontal y vertical marcadas.

Tabla 7.21. Métodos de la clase *CardLayout*

Método	Descripción
void addLayoutComponent(Component comp, Object constraints)	Añade el componente al *layout*, usando el objeto dado.
void addLayoutComponent(String nombre, *Component comp)*	Añade al *layout*el componente dado, con el nombre indicado.
void first(Container padre)	Va a la primera tarjeta del contenedor.
int getHgap()	Obtiene la separación horizontal entre componentes.
float getLayoutAlignmentX(Container padre)	Obtiene la alineación a lo largo del eje x.
float getLayoutAlignmentY(Container padre)	Obtiene la alineación a lo largo del eje y.
int getVgap()	Obtiene la separación vertical entre componentes.

Método	Descripción
void invalidateLayout(Container target)	Invalida el esquema.
void last(Container padre)	Va a la última tarjeta del contenedor.
void layoutContainer(Container padre*)*	Pone el contenedor dado usando este *card layout.*
Dimension maximumLayoutSize (Container target)	Obtiene las dimensiones máximas del esquema dados los componentes en el *target.*
Dimension minimumLayoutSize (Container padre*)*	Calcula el tamaño mínimo para el panel dado.
void next(Container padre)	Va a la siguiente tarjeta del contenedor dado.
Dimension preferredLayoutSize (Container padre*)*	Obtiene las dimensiones preferidas del esquema usando este *card layout.*
void previous(Container padre)	Va a la tarjeta anterior del contenedor dado.
void removeLayoutComponent (Component comp)	Elimina del esquema el componente dado.
void setHgap(int hgap)	Establece la separación horizontal entre los componentes.
void setVgap(int vgap)	Establece la separación vertical entre los componentes.
void show(Container padre, *String* nombre)	Va al componente que se añadió al esquema con el nombre especificado.
String toString()	Obtiene una representación tipo *string* del esquema.

Veamos un ejemplo en el que se crea una clase panel, *cardPanel*, que visualiza un botón, sobre el que el usuario puede hacer clic para moverse a la siguiente tarjeta, y una etiqueta para indicar el número de la tarjeta actual. El constructor de esta clase tiene dos parámetros: un objeto de la clase principal de *applet*, por lo que la *applet*, en sí misma, puede gestionar los clics sobre el botón que se produzcan en cada panel y una representación en cadena del número del panel actual:

```
import java.awt.*;
import java.applet.Applet;
```

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

/*
<APPLET
    CODE=card.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

class cardPanel extends Panel
{
    Button button;
    Label label;

    cardPanel(card applet, String cardnumber)
    {
        button = new Button("Tarjeta siguiente");
        button.addActionListener(applet);
        add(button);
        label = new Label("Esta es la tarjeta nº " + cardnumber);
        add(label);
    }
}
```

En la clase *main* de la *applet,* se crean tres paneles de la clase *cardPanel.* Después se les añade a un *card layout,* dándoles los nombres "primera", "segunda", y "tercera":

```
public class card extends Applet implements ActionListener
{
    int index = 1;
    CardLayout cardlayout;
    cardPanel panel1, panel2, panel3;

    public void init()
    {
        cardlayout = new CardLayout();
        setLayout(cardlayout);

        panel1 = new cardPanel(this, "uno");
        panel2 = new cardPanel(this, "dos");
        panel3 = new cardPanel(this, "tres");

        add("primero", panel1);
        add("segundo", panel2);
        add("tercero", panel3);

        cardlayout.show(this, "primero");
    }
```

Cuando el usuario hace clic sobre un botón, se puede repetir sobre las tarjetas disponibles, usando un índice que se incrementa y se evalúa con la sentencia *switch:*

```
    public void actionPerformed(ActionEvent event)
    {
        switch(++index){
            case 1:
                cardlayout.show(this, "primero");
                break;
            case 2:
                cardlayout.show(this, "segundo");
                break;
            case 3:
                cardlayout.show(this, "tercero");
                break;
        }
        if(index == 3) index = 0;
        repaint();
    }
}
```

El resultado se muestra en la figura 7.11. Cuando el usuario hace clic sobre el botón "Tarjeta siguiente", la *applet* cambia a la siguiente tarjeta del esquema, dando vueltas sobre las tarjetas. Veamos un *card layout* y cómo funciona; esta *applet* se puede encontrar en *card.java* en el CD.

Hay un esquema más que vamos a revisar, *grid bag layouts*.

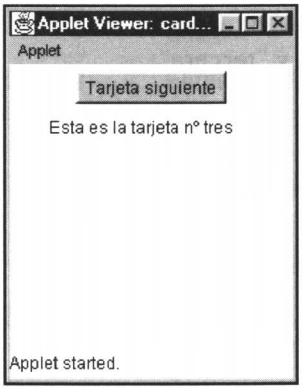


Figura 7.11. Usar un *card layout*.

Grid bag layouts

"De acuerdo", dice el programador novato. "Ya soy un experto en los esquemas de AWT". "No, todavía no", le dice. "No hasta que sea un experto en *grid bag layout*".

Grid bag layouts son los esquemas AWT más complejos y flexibles, y permiten especificar, más exactamente que cualquier otro gestor de esque-

mas, el lugar donde se quiera que vayan los componentes. Este es el diagrama de herencia para la clase *GridBagLayoutClass*.

```
java.lang.Object
|____java.awt.GridBagLayout
```

El constructor de *GridBagLayoutClass* se encuentra en la tabla 7.22 y sus métodos en la tabla 7.23.

Tabla 7.22. Constructor de la clase *GridBagLayout*.

Constructor	Descripción
GridBagLayout()	Crea un gestor de *grid bag layout.*

Tabla 7.23. Métodos de la clase *GridBagLayout*.

Método	Descripción
void addLayoutComponent(Component comp, Object constraints)	Añade el componente dado al esquema, usando el objeto *constraint.*
void addLayoutComponent(String nombre, *Component comp)*	Añade al esquema el componente dado.
protected void AdjustForGravity (GridBagConstraints constraints, Rectangle r)	Ajusta las características de gravedad.
protected void ArrangeGrid(container padre)	Obtiene la cuadrícula del padre.
GridBagConstraints getConstraints (Component comp)	Obtiene los *constraints* del componente dado.
float getLayoutAlignmentX(Container padre)	Se obtiene la alineación a lo largo del eje x.
float getLayoutAlignmentY(Container padre)	Se obtiene la alineación a lo largo del eje y.
int[][] getLayoutDimensions()	Determina la anchura de la columna y las alturas de la fila para el *layout grid.*
protected java.awt.GridBagLayoutInto GetLayoutInfo(Container padre, int sizeflag)	Obtiene el *layout constraints.*
Point getLayoutOrigin()	Determina el origen del *layout grid.*

Método	Descripción
double[][] getLayoutWeights()	Determina los pesos de las filas y columnas del *layout grid.*
protected Dimension GetMinsize (Container padre, java.awt.GridBag LayoutInfo info)	Obtiene el tamaño mínimo del contenedor.
void invalidateLayout(Container target)	Invalida el esquema.
void layoutContainer(Container padre)	Pone el contenedor.
Point location(int x, int y)	Determina qué celda del *layout* contiene un punto.
protected GridBagConstraints lookupConstraints(Component comp)	Recupera los *constraints* para un componente dado.
Dimension maximumLayoutSize (Container target)	Obtiene las dimensiones máximas del esquema dados los componentes del contenedor.
Dimension minimumLayoutSize (Container padre)	Determina el tamaño mínimo del contenedor, usando este esquema.
Dimension preferredLayoputSize (Container padre)	Determina el tamaño preferido del contenedor, usando el esquema.
void removeLayoutComponent (Component comp)	Retira del esquema, el componente dado.
void setConstraints(Component comp, GridBagConstraints constraints)	Establece los *constraints* para los componentes dados en el esquema.
String toString()	Obtiene una representación tipo *string* de este esquema.

Añadir componentes a un contenedor usando un *grid bag layout* se hace igual que con un *grid layout*, salvo que se tienen más opciones; por ejemplo, se pueden poner las alturas y anchuras relativas de los componentes. La inicialización de un *grid bag layout* es un proceso de dos pasos: primero se configura la posición relativa de un componente respecto a los otros, y luego se añade el componente al esquema.

Los componentes se configuran utilizando la clase *GridBagConstraints*; los campos de esa clase aparecen en la tabla 7.24, sus constructores en la tabla 7.25 y su método en la tabla 7.26.

Tabla 7.24. Campos de la clase *GridBagConstraints*.

Campo	Descripción
int anchor	Usado cuando el componente es más pequeño que su área de visualización.
static int BOTH	Redimensiona el componente horizontal y verticalmente.
static int CENTER	Pone el componente en el centro de su área de visualización.
static int EAST	Pone el componente en el lado derecho de su área de visualización, centrado verticalmente.
int fill	Este campo se usa cuando el área de visualización del componente es más grande que el tamaño solicitado por él.
int gridheight	Indica el número de celdas en una columna para el área de visualización del componente.
int gridwidth	Indica el número de celdas en una fila para el área de visualización del componente.
int gridx	Indica la celda a la izquierda del área de visualización del componente.
int gridy	Indica la celda en la parte superior del área de visualización del componente .
static int horizontal	Redimensiona el componente, horizontalmente.
Insets insets	Indica el *padding* externo del componente.
int ipadx	Indica el *padding* interno x del componente.
int ipady	Indica el *padding* interno y del componente.
static int NONE	Indica que no se redimensione el componente.
static int NORTH	Pone el componente en la parte superior de su área visualizable,centrada horizontalmente.
static int NORTHEAST	Pone el componente en la esquina superior derecha de su área de visualización.
static int NORTHWEST	Pone el componente en la esquina superior derecha de su área de visualización.
static int RELATIVE	Indica que este componente es el siguiente al último componente en su columna o fila, o que

Campo	Descripción
	este componente está situado a continuación del componente previamente añadido.
static int REMAINDER	Especifica que este componente es el último en su columna o fila.
static int SOUTH	Pone el componente en el fondo de su área visualizable, centrado horizontalmente.
static int SOUTHEAST	Pone el componente en la esquina inferior izquierda de su área de visualización.
static int SOUTHWEST	Pone el componente en la esquina inferior izquierda de su área de visualización.
static int VERTICAL	Redimensiona el componente verticalmente, pero no horizontalmente.
double weightx	Especifica cómo se distribuye el espacio horizontal extra.
double weighty	Especifica cómo se distribuye el espacio vertical extra.
static int WEST	Pone el componente en el lado izquierda de su área de visualización, centrado verticalmente.

Tabla 7.25. Constructores de la clase *GridBagConstraints*.

Constructor	Descripción
GridBagConstraints()	Crea un objeto *GridBagConstraint.*
GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, insets, insets, int ipadx, int ipady)	Crea un objeto *GridBagConstraints*con todos los campos inicializados con los valores pasados.

Tabla 7.26. Método de la clase *GridBagConstraints*.

Método	Descripción
Object clone()	Crea una copia de este *grid bag constraint.*

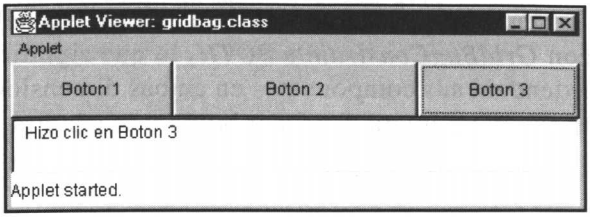


Figura 7.12. Usar un *grid bag layout.*

Veamos un ejemplo de cómo funciona un *grid bag layout.* Crearemos la *applet* que se muestra en la figura 7.12; aquí, se han puesto tres botones en la fila superior del esquema, haciendo que el botón del medio sea el doble que los otros, y se añade un cuadro de texto debajo. Cuando el usuario hace clic sobre un botón, la *applet* devuelve el botón que se ha pulsado, como se puede ver.

Se puede especificar las coordenadas x e y para los componentes que se añadan a un *grid bag layout* usando los campos *weightx* y *weighty*. Estos valores representan los pesos relativos que se quieren dar a los componentes en la misma fila o columna. En este caso, se dará a cada componente el mismo peso y, pero al segundo botón se le da el doble de peso x que a los otros.

Comencemos creando un *grid bag layout* y las restricciones de un objeto:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=gridbag.class
    WIDTH=400
    HEIGHT=80 >
</APPLET>
*/

public class gridbag extends Applet implements ActionListener
{
    Button button1, button2, button3;
    TextField text1;

    public void init()
    {
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        setLayout(gridbag);
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.BOTH;
        .
        .
        .
```

Observe que el campo *fill* de *constraints* del objeto se está inicializando con *GridBagConstraints.BOTH*, lo que significa que el gestor de esquemas extenderá sus componentes en ambas dimensiones para rellenarlas. A continuación, se creará el primer botón y se añadirá al esquema, como sigue:

```
public void init()
    {
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        setLayout(gridbag);
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.BOTH;

        constraints.weigthx = 1;
        button1 = new Button("Botón 1");
        gridbag.setConstraints(button1, constraints);
        button1.setActionCommand("Botón 1");
        add(button1);
        button1.addActionListener(this);
        .
        .
        .
```

El primer botón tiene un peso x de 1, pero al siguiente botón se le dará un peso x de 2, para hacer que sea el doble de ancho que los otros dentro de la misma fila:

```
public void init()
    {
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        setLayout(gridbag);
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.BOTH;

        constraints.weightx = 1;
        button1 = new Button("Botón 1");
        gridbag.setConstraints(button1, constraints);
        button1.setActionCommand("Botón 1");
        add(button1);
        button1.addActionListener(this);

        constraints.weightx = 2;
        button2 = new Button("Botón 2");
        gridbag.setConstraints(button2, constraints);
        button2.setActionCommand("Botón 2");
        add(button2);
        button2.addActionListener(this);

        constraints.weightx = 1;
        button3 = new Button("Botón 3");
        constraints.gridwidth = GridBagConstraints.REMAINDER;
        gridbag.setConstraints(button3, constraints);
```

```
        button3.setActionCommand("Botón 3");
        add(button3);
        button3.addActionListener(this);

        text1 = new TextField();
        constraints.gridwidth = GridBagConstraints.REMAINDER;
        gridbag.setConstraints(text1, constraints);
        add(text1);
    }
```

Lo único que queda es gestionar los clics sobre el botón y hacer que se visualice el número del botón sobre el que se ha hecho clic:

```
public void actionPerformed(ActionEvent e)
    {
        text1.setText("Hizo clic en " +
            ((Button) e.getSource()).getActionCommand());
    }
```

Y eso es todo. Ahora este código creará la *applet* que se muestra en la figura 7.12. Este ejemplo se puede encontrar en el fichero *gridbag.java* en el CD.

Usar intercalados y rellenos

"Veamos", dice el programador novato, "¿hay alguna forma de añadir un borde alrededor de los componentes del esquema?" "Claro que la hay", le responde. "Se pueden usar los intercalados y los rellenos".

Con los intercalados se puede añadir espacio entre los bordes del contenedor y los componentes. Estos intercalados se crean con la clase *Insets,* que tiene el siguiente constructor:

```
Insets (int top, int left, int bottom, int right)
```

Este es un ejemplo en el que se añaden intercalados de 20 pixels en todos los bordes de la *applet* desarrollada en el apartado anterior:

```
public Insets getInsets()
{
   return new Insets(20, 20, 20, 20);
}
```

El resultado se muestra en la figura 7.13. Como se puede ver en esta figura, hay un espacio de 20 pixels alrededor de los componentes del contenedor.

Además se pueden rellenar los componentes individuales. Esto se puede hacer con los métodos *setHgap* y *setVgap* de los *card layout, flow layout* y

grid layout, así como usando los miembros *ipadx* e *ipady* de los *grid bag layouts*. Por ejemplo, se puede modificar la calculadora de multiplicar que se escribió anteriormente en este capítulo para añadir un espacio vertical de 10 pixels entre los componentes, como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=multiplier2.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class multiplier2 extends Applet implements ActionListener {

    TextField text1, text2, answertext;
    Label multiplylabel;
```

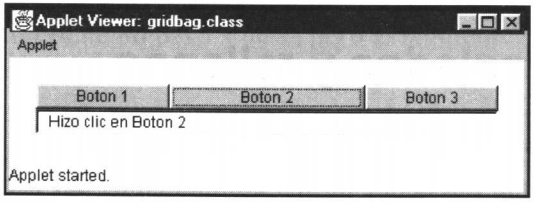


Figura 7.13. Usar intercalados.

```
Button button1;
    GridLayout g;

    public void init()
    {
        g = new GridLayout(5, 1);
        g.setVgap(10);

        setLayout(g);

        text1 = new TextField(10);
        add(text1);

        multiplylabel = new Label("*", Label.CENTER);
        add(multiplylabel);

        text2 = new TextField(10);
        add(text2);

        button1 = new Button("=");
        add(button1);
```

```
        button1.addActionListener(this);

        answertext = new TextField(10);
        add(answertext);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == button1){
            int product = Integer.parseInt(text1.getText()) *
                Integer.parseInt(text2.getText());
            answertext.setText(String.valueOf(product));
        }
    }
}
```

El resultado se muestra en la figura 7.14.

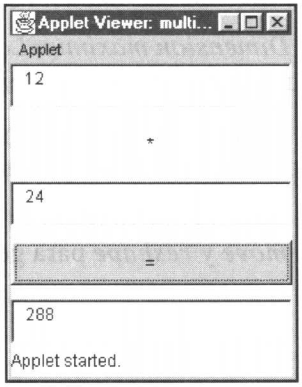


Figura 7.14. Usar relleno.

Crear el propio gestor de esquemas

Uno mismo puede poner los componentes en un contenedor si desinstala el gestor de esquemas del contenedor (ver la introducción de este capítulo para más detalles). De hecho, podrá incluso crear su propio gestor de esquemas, si se implementa la interfaz *LayoutManager2* en una clase (la interfaz *LayoutManager* no soporta las restricciones). Estos son los métodos de *LayoutManager2* que se tendrán que sobrescribir:

- *public void addLayoutComponent(String* nombre, *Component* componente): este método añade un componente al esquema. La clase contenedor le llama, sobrescribe este método si se quieren saber los nombres de los componentes del esquema.

- *public void removeLayoutComponent(Component* componente): elimina un componente del esquema.
- *public float getLayoutAlignmentX(Container target)* and *public float getLayoutAlignmentX(Container target):* Especifica cómo alinear los componentes en las direcciones x e y, devolviendo un valor de 0 al alinear los componentes en el origen, devolviendo 1 al alinearlo tan lejos como se pueda del origen y devolviendo 0.5 al centrar el componente.
- *public void invalidateLayout(Container target):* Invalida un esquema, lo que significa que se debería borrar cualquier información oculta si se llama a este método.
- *public Dimension preferredLayoutSize(Container* padre): Devuelve el tamaño ideal del contenedor padre.
- *public Dimension maximumLayoutSize(Container target):* Devuelve el tamaño máximo del componente.
- *public Dimension minimuLayoutSize(Container target):* Devuelve el tamaño mínimo del componente.
- *public void layoutContainer(Container* padre): Utiliza los métodos *resize, move* y *reshape* para gestionar los componentes.

8 AWT: Listas, cuadros de lista, áreas de texto, barras y cuadros de desplazamiento

Este capítulo trata un número importante de componentes AWT: listas, cuadros de lista desplegables, cuadros de texto, barras de desplazamiento y cuadros de desplazamiento. Estos componentes son fundamentales para la programación AWT, resultan familiares para casi todos los usuarios GUI, y los veremos rápidamente en esta sección. Observe que uno de los temas principales de este capítulo es el desplazamiento; todos los controles de este capítulo lo soportan de una u otra forma, permitiendo al usuario moverse, fácilmente, por grandes documentos.

Listas

Como su nombre indica, un control de tipo lista presenta al usuario una lista de elementos, cadenas de texto, en la que puede hacer una selección. En los entornos de trabajo con ventanas, con frecuencia, el espacio es un premio, por lo que es una buena idea colocar elementos en listas, ya que usando las barras de desplazamiento, se pueden tener listas largas en componentes de

lista cortos. El usuario puede seleccionar el elemento de la lista que quiera e iniciar alguna acción, haciendo doble clic sobre él. Además, las listas pueden soportar la selección múltiple, usando las teclas Mayús y Control. Sin embargo, si se soporta la selección múltiple, hay que pensar en cómo utilizar el ratón dentro de una lista; porque al hacer clic sobre un elemento se selecciona y luego, con el doble clic, se inicia algún tipo de acción. Al hacer doble clic sobre un elemento, el resto se deselecciona. Para resolver este problema, *Sun* sugiere el uso de algún otro evento, como puede ser hacer clic sobre un botón, para iniciar una acción cuando se trata de selecciones múltiples.

Cuadros de lista desplegables

Los cuadros de lista desplegables presentan al usuario una lista de elementos para seleccionar, pero hay una diferencia, son más compactos que las listas. Los cuadros de lista desplegables se parecen a los cuadros de texto, aunque no se pueden editar, con un pequeño botón a la derecha que tiene una flecha apuntando hacia abajo. Cuando el usuario hace clic sobre la flecha, se abre una lista que contiene todas las selecciones disponibles, y el usuario puede seleccionar una de ellas. Una vez hecha la selección, se cierra la lista y se visualiza la selección actual. Si el usuario abre otra vez la lista, la selección actual aparece resaltada. Observe que estos cuadros sólo están diseñados para visualizar al mismo tiempo una única selección, lo que significa que no se puede seleccionar a la vez más de un elemento. Usando las teclas Mayús o Control y haciendo clic al mismo tiempo se obtiene el mismo efecto que si se hace clic en un elemento.

Como el resto de componentes de este capítulo, los cuadros de listas desplegables soportan el desplazamiento. Si la lista de elementos es larga, las barras de desplazamiento aparecerán al lado de la lista automáticamente.

Áreas de texto

Las áreas de texto son como los cuadros de texto, salvo que soportan texto en dos dimensiones, con filas y columnas, por lo que se puede visualizar más texto. En el momento de crear el área de texto, hay que indicar su tamaño, en filas y columnas (medido en caracteres). El resultado es un cuadro de texto que tiene el número de filas y columnas que se haya especificado. Cuando se quiere trabajar con documentos enteros, en vez de con líneas de texto, se usan las áreas de texto.

Al igual que el resto de los componentes de este capítulo, las áreas de texto soportan el desplazamiento. Se puede especificar si se quiere una barra de desplazamiento horizontal, vertical, ambas o ninguna. Si no se tiene una barra de desplazamiento horizontal (y sólo si no tiene una barra de desplazamiento horizontal), la opción de ajuste de palabras se habilita automáticamente.

Barras de desplazamiento

Las barras de desplazamiento son, por supuesto, los elementos de desplazamiento más importantes, y todos los usuarios de interfaces gráficas conocen su existencia y su funcionamiento. Los usuarios utilizan el ratón para manipular el *indicador* (también denominado cuadro de desplazamiento, elevador o burbuja) de la barra de desplazamiento seleccionado un valor dentro de un intervalo continuo. Este intervalo se establece al crear la barra de desplazamiento y, cuando el usuario selecciona un valor diferente, podemos capturar los eventos generados por la barra de desplazamiento para leer este nuevo valor. El usuario también puede hacer clic sobre los dos botones de flecha que hay en los extremos de la barra de desplazamiento para aumentar o disminuir el valor en cantidades fijas que habremos establecido previamente. Por último, el usuario puede hacer clic sobre el recorrido de la propia barra de desplazamiento (es decir, la zona comprendida entre el cuadro de desplazamiento y los botones de desplazamiento) para aumentar o disminuir el valor en cantidades fijas (normalmente mayores que las anteriores) que también habremos establecido previamente.

Paneles de desplazamiento

Es posible que se quieran utilizar las barras de desplazamiento para moverse en otros componentes, como puede ser un cuadro de texto largo; los controles de Java las soportan pero no del todo. Para hacer que el desplazamiento en otros componentes sea más fácil, Java tiene la clase *ScrollPane*. Se usan las áreas de desplazamiento para añadirles componentes y permitir que el área de desplazamiento los gestione. Una vez que se haya añadido un gran componente a un área de desplazamiento, de una vez, sólo será visible una parte de él, y se podrán usar las barras de desplazamiento del área para ir a otras partes de ese componente. Eso es todo. Ahora que ya hemos revisado el contenido de este capítulo, es el momento de pasar a la siguiente sección.

Usar las áreas de texto

"Uh-oh", dice el programador novato, "tengo un problema. He escrito mi novela en un cuadro de texto de Java, y..." Usted sonríe y le dice, "¿Y se ha quedado sin espacio?" "Claro", dice el programador novato, "no es muy conveniente escribir una novela entera como si fuera una línea de texto". "Pruebe las áreas de texto", le sugiere.

Un área de texto es un cuadro de texto de dos dimensiones; de hecho, tienen una gran ventaja, se pueden visualizar todos los documentos en áreas de texto, incluyendo el tratamiento de las palabras al final de las líneas. Además se pueden usar las barras de desplazamiento para moverse por el texto. Observe, sin embargo, que si ha habilitado una barra de desplazamiento horizontal, este tratamiento de las palabras al final de la línea quedará deshabilitado. Este es el diagrama de herencia de la clase AWT *TextArea*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.TextComponent
          |____java.awt.TextArea
```

Verá los campos de la clase *TextArea*, que se usan en los constructores de la clase *TextArea*, en la tabla 8.1, sus constructores en la tabla 8.2 y sus métodos en la 8.3.

Veamos un ejemplo. En este caso, se creará un área de texto de 10 filas y 20 columnas (observe que estas dimensiones se miden en caracteres). Además se añadirá un botón sobre el que el usuario puede hacer clic para visualizar el texto "¡Hola desde Java!" en el área de texto.

Tabla 8.1. Campos de la clase *TextArea*.

Campo	Descripción
static int SCROLLBARS_BOTH	Añade las barras de desplazamiento horizontal y vertical.
static int SCROLLBARS_HORIZONTAL_ONLY	Añade, únicamente, una barra de desplazamiento horizontal.
static int SCROLLBARS_NONE	Indica que no se creará ninguna barra de desplazamiento en el área de texto.
static int SCROLLBARS_VERTICAL_ONLY	Sólo visualiza una barra de desplazamiento vertical.

Tabla 8.2. Constructores de la clase *TextArea*.

Constructor	Descripción
TextArea()	Construye una nueva área de texto.
TextArea(int filas, *int* columnas)	Construye una nueva área de texto, vacía, con el número de filas y columnas indicado.
TextArea(String texto*)*	Construye una nueva área de texto con el texto dado.
TextArea(String texto, *int* filas, *int* columnas)	Construye una nueva área de texto, con el texto dado y con el número de filas y de columnas dado.
TextArea(String texto, *int* filas, *int* columnas, *int* barras_de_desplazamiento)	Construye una nueva área de texto con el texto dato y fija la visibilidad de las filas, columnas y barras de desplazamiento.

Tabla 8.3. Métodos de la clase *TextArea*.

Método	Descripción
void addNotify()	Crea el compañero del área de texto.
void appendText(String str)	Añade texto al texto actual del área de texto.
void appendText(String str)	Obsoleto. Reemplazado por *append(String)*.
int getColumns()	Obtiene el número de columnas del área de texto.
Dimension getMinimumSize()	Establece el tamaño mínimo del área de texto.
Dimension getMinimumSize(int filas, *int* columnas)	Fija el tamaño mínimo del área de texto con el número de filas y de columnas dado.
Dimension getPreferredSize()	Establece el tamaño preferido del área de texto.
Dimension getPreferredSize(int filas, *int* columnas)	Establece el tamaño preferido de un área de texto con el número de filas y de columnas indicado.

Método	Descripción
int getRows()	Obtiene el número de filas del área de texto.
int getScrollbarVisibility()	Obtiene un valor enumerado que indica la visibilidad de la barra de desplazamiento.
void insert(String str, int pos)	Inserta el texto dado en la posición del área de texto indicada.
void insertText(Sring str, int pos)	Obsoleto. Reemplazado por *insert-(String, int).*
Dimension minimumSize()	Obsoleto. Reemplazado por *getMini-mumSize().*
Dimension minimumSize(int filas, *int* columnas)	Obsoleto. Reemplazado por *getMini-mumSize(int, int).*
protected String paramString()	Devuelve una representación en cadena del estado del área de texto.
Dimension preferredSize()	Obsoleto. Reemplazado por *getPre-ferredSize().*
Dimension preferredSize(int filas, *int* columnas)	Obsoleto. Reemplazado por *getPre-ferredSize(int, int).*
void replaceRange(String str, int inicio, *int* final)	Cambia el texto entre las posiciones de inicio y final.
void replaceText(String str, int inicio, *int* final)	Obsoleto. Reemplazado por *replace-Range(String, int, int).*
void setColumns(int columnas)	Fija el número de columnas.
void setRows(int filas)	Fija el número de filas.

Hay varias formas de situar texto en un área de texto. Por ejemplo, se puede usar el método *insert* para poner texto en una posición específica (comenzando en 0) de la cadena que el área de texto gestiona, o se puede usar el método *append* para añadir texto a continuación del actual. Así es cómo se usa el método *insert:*

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
```

```
<APPLET
    CODE=areadetexto.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/
public class areadetexto extends Applet implements ActionListener {

    TextArea textarea1;
    Button button1;
    public void init()
    {
       textarea1 = new TextArea("", 10, 20, TextArea.SCROLLBARS_BOTH);
        add(textarea1);
        button1 = new Button("¡Haga clic aquí!");
        add(button1);
        button1.addActionListener(this);
    }

    public void actionPerformed (ActionEvent e)
    {
        String msg = "¡Hola desde Java!";
        if(e.getSource() == button1){
            textarea1.insert(msg, 0);
        }
    }
}
```

Además, observe que se han añadido al área de texto barras de desplazamiento horizontal y vertical. El resultado aparece en la figura 8.1. Con las áreas de texto se pueden hacer más cosas, por ejemplo, se puede seleccionar y reemplazar texto. Ver el siguiente apartado para más detalles.

Reemplazar texto en áreas de texto

El especialista en soporte de productos (ESP) no está contento, como es usual, y le dice, "Los usuarios se están quejando del nuevo procesador de textos que ha escrito en Java". "¿Sí?" pregunta. "Quieren que sea posible seleccionar y reemplazar texto", dice ESP. "Algunas personas", dice, "nunca están satisfechas". Se puede usar el método *replaceRange* para reemplazar un rango de texto en un área de texto:

```
void replaceRange(String str, int start, int end);
```

Donde, *str* es la cadena con la que se quiere reemplazar el texto antiguo, *start* es el comienzo del rango de caracteres a reemplazar y *end* es el final del rango.

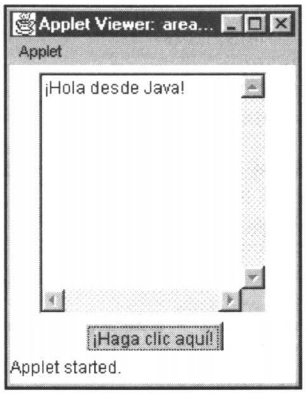


Figura 8.1. Usar un área de texto.

Este es un ejemplo en el que se visualiza el texto "Ya es la hora." en un área de texto. El usuario puede seleccionar parte o todo de ese texto y hacer clic sobre un botón para reemplazar el texto seleccionado con el mensaje "¡Hola desde Java!". Empezamos creando la nueva área de texto e inicializándola con el texto "Ya es la hora.":

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=reemplazar.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class reemplazar extends Applet implements ActionListener
{
    TextArea textarea1;
    Button button1;
    public void init()
    {
        textarea1 = new TextArea("Ya es la hora.", 5, 20,
            TextArea.SCROLLBARS_BOTH);
        add(textarea1);
        button1 = new Button("Cambiar el texto seleccionado");
        add(button1);
        button1.addActionListener(this);
    }
    .
    .
    .
}
```

Cuando el usuario hace clic sobre el botón, se puede determinar el texto seleccionado utilizando los métodos *getSelectionStart* y *getSelectionEnd* y luego reemplazar el texto seleccionado con "¡Hola desde Java!", como sigue:

```
public void actionPerformed (ActionEvent e){
        if(e.getSource() == button1){
            textarea1.replaceRange("¡Hola desde Java!",
                textarea1.getSelectionStart(),
                textarea1.getSelectionEnd()) ;
        }
```

El resultado se muestra en la figura 8.2, y encontrará este ejemplo en el CD como reemplazar.java. Además de reemplazar el texto, se puede buscar. Ver el siguiente apartado.

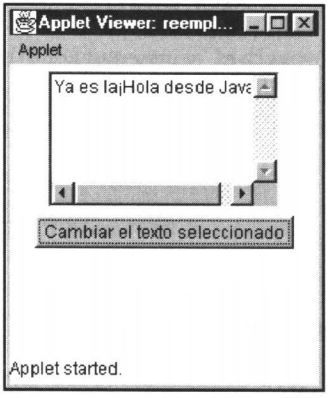


Figura 8.2. Reemplazar el texto seleccionado en un área de texto.

Buscar y seleccionar texto en áreas de texto

El especialista en soporte de productos regresa descontento de nuevo. "Ahora, los usuarios quieren poder buscar texto cuando usan su procesador de textos", informa. "¿Qué hacemos?", le pregunta, exasperado. "¿Verificar letra por letra?". La clase *TextArea* no soporta ningún método directo para buscar texto pero la clase *String* sí. Por lo tanto, se puede copiar el texto de un área de texto en un objeto *String* y buscar ese objeto en el texto que se quiera, usando el método *indexOf*. Este es un ejemplo en el que se permite al usuario hacer clic sobre un botón para buscar y seleccionar la palabra hora en el texto "Ya es la hora." de un área de texto. Empezaremos creando una nueva área de texto, y luego lo inicializaremos con el texto "Ya es la hora.":

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/*
<APPLET
    CODE=buscar.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/
public class buscar extends Applet implements ActionListener
{
    TextArea textarea1;
    Button button1;
    public void init()
    {
            textarea1 = new TextArea("Ya es la hora.", 5, 20,
TextArea.SCROLLBARS_BOTH);
        add(textarea1);
        button1 = new Button("Buscar \"hora\"");
        add(button1);
        button1.addActionListener(this);
    }
    .
    .
    .
}
```

Ahora, cuando el usuario hace clic sobre el botón, el texto del área se carga en una cadena llamada s:

```
public void actionPerformed (ActionEvent e){
        if(e.getSource() == button1){
            String s = textarea1.getText();
            .
            .
            .
        }
}
```

Ahora, se puede buscar el texto "hora" usando el método *indexOf* de la clase *String*:

```
public void actionPerformed (ActionEvent e){
        if(e.getSource() == button1){
            String s = textarea1.getText();
            String s2 = new String("hora");

            int location = s.indexOf(s2);
            .
            .
            .
        }
}
```

Una vez que se haya encontrado la cadena "hora", se usará el método *TextAreaselect* (heredado de la clase AWT *TextComponent*) para resaltar el texto. Así es como se usa *select* en general:

```
void select (int selectionStart, int selectionEnd);
```

Así es como usaríamos el método *select* en el código:

```
public void actionPerformed (ActionEvent e){
        if(e.getSource() == button1){
            String s = textarea1.getText();
            String s2 = new String("hora");
            int location = s.indexOf(s2);
            textarea1.select(location, location + s2.length()) ;
         }
}
```

El resultado de este código aparece en la figura 8.3. Cuando el usuario hace clic sobre el botón, el código busca y resalta el texto "hora" en el área de texto. Este ejemplo se llama buscar.java en el CD.



Figura 8.3. Buscar y seleccionar texto en un área de texto.

Usar listas

El especialista de soporte de productos regresa y dice, "Hay un problema". "¿Sí?", le pregunta. "Es sobre su último programa de catálogo de CD de música clásica, hay más de cien mil listas". "Correcto", le dice prudentemente. "Por supuesto, la lista es más larga que la pantalla". "Aproximadamente dos mil pies de larga", le dice ESP.

Si se tiene una lista larga de elementos para presentar, hay que pensar en utilizar un control de tipo lista. Se puede rellenar una lista con muchos elementos de texto, usando el método *add* de la clase *List* y presentar al usuario sólo unos pocos al mismo tiempo (se puede seleccionar cuántos). El usuario puede resaltar un elemento de la lista haciendo clic sobre él y puede hacer doble clic para iniciar alguna acción. De hecho, el usuario puede seleccionar múltiples elementos de una lista e iniciar alguna acción, normalmente haciendo clic sobre un botón. Este es el diagrama de herencia de la clase *List*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.List
```

Los constructores de la clase *List* se encuentran en la tabla 8.4 y sus métodos en la tabla 8.5. Hay que prestar especial atención a la tabla de métodos; muchos de ellos son útiles, como *getItemCount,* que devuelve el número de elementos de la lista.

Tabla 8.4. Constructores de la clase *List.*

Constructor	Descripción
List()	Crea un nuevo cuadro de desplazamiento.
List(int filas)	Crea un nuevo cuadro de desplazamiento, con el número de líneas visibles que se indica.
List (int filas, *boolean multipleMode)*	Crea una nueva lista con el número de filas especificado y se habilita la selección múltiple, si *multipleMode* es verdadero.
void add(String elemento)	Añade el elemento dado al final de la lista.
void add(String elemento, *int* índice)	Añade el elemento dado a la lista en la posición marcada por el índice.
void addActionListener(ActionListener l)	Añade el *action listener* dado para obtener los eventos de acción de la lista.
void addItem(String elemento)	Obsoleto. Reemplazado por *add (String).*

Constructor	Descripción
void addItem(String elemento, *int* índice)	Obsoleto. Reemplazado por *add (String, int).*
void addItemListener(itemListener l)	Añade el *item listener* dado para obtener eventos de elemento de la lista.
void addNotify()	Crea el compañero de la lista.
boolean allowsMultipleSelections()	Obsoleto. Reemplazado por *isMultipleMode().*
void clear()	Obsoleto. Reemplazado por *removeAll().*
int countItems()	Obsoleto. Reemplazado por *getItemCount().*
void delItem(int position)	Obsoleto. Reemplazado por *remove (String)* y *remove(int).*
void delItem(int start, int end)	Obsoleto. No es de uso público.
void deselect(int índice)	Deselecciona el elemento asociado al índice dado.
String getItem(int índice)	Obtiene el elemento asociado al índice dado.
int getItemCount()	Obtiene el número de elementos de la lista.
String[] getItems()	Obtiene los elementos de la lista.
Dimension getMinimumSize()	Determina el tamaño mínimo de la lista.
Dimension getMinimumSize(int filas)	Obtiene las dimensiones mínimas de la pantalla para una lista con el número de filas dado.
Dimension getPreferredSize()	Obtiene el tamaño preferido de esta lista.
Dimension getPreferredSize(int filas)	Obtiene las dimensiones preferidas para una lista con el número de filas dado.
int getRows()	Obtiene el número de líneas visibles de la lista.

Constructor	Descripción
int getSelectedIndex()	Obtiene el índice del elemento seleccionado en la lista.
int[] getSelectedIndexes()	Obtiene los índices seleccionados de la lista.
String getSelectedItem()	Obtiene el elemento seleccionado de esta lista.
String[] getSelectedItems()	Obtiene los elementos seleccionados de esta lista.
Object[] getSelectedObjects()	Obtiene los elementos seleccionados de la lista en un *array* de objetos.
int getVisibleIndex()	Obtiene el índice del último elemento que se hizo visible con el método *makeVisible.*
boolean isIndexSelected(int índice)	Determina si el elemento dado de la lista está seleccionado.
boolean isMultipleMode()	Determina si la lista permite hacer selecciones múltiples.
boolean isSelected(int índice)	Obsoleto. Reemplazado por *is IndexSelected(int).*
void makeVisible(int índice)	Hace que el elemento del índice dado sea visible al principio de la lista.
Dimension minimumSize()	Obsoleto. Reemplazadopor *getMinimumSize().*
Dimension minimumSize(int filas)	Obsoleto. Reemplazado por *getMinimumSize().*
protected String paramString()	Obtiene el *string* que representa el estado de esta lista.
Dimension preferredSize()	Obsoleto. Reemplazado por *getPreferredSize().*
Dimension preferredSize(int filas)	Obsoleto. Reemplazado por *getPreferredSize(int).*
protected void processActionEvent (ActionEvent e)	Procesa los eventos de acción que ocurren en este componente envián-

Constructor	Descripción
	dolos a los objetos *ActionListener* registrados.
protected void processEvent (AWTEvent e)	Procesa los eventos de esta lista.
protected void processItemEvent (ItemEvent e)	Procesa eventos de elemento que se produzcan en la lista enviándolos a los objetos *ItemListener* registrados.
void remove(int posición)	Elimina el elemento de la lista que está en la posición dada.
void remove(String elemento)	Elimina la primera ocurrencia de un elemento de la lista.
void removeActionListener(ActionListener l)	Elimina el *action listener* dado, para que no reciba más eventos de acción de la lista.
void removeAll()	Elimina todos los elementos de la lista.
void removeItemListener(ItemListener l)	Elimina el *item listener* dado para que no reciba más eventos de elemento de la lista
void removeNotify()	Elimina el compañero de la lista.
void replaceItem(String valor_nuevo, *int* índice)	Reemplaza el elemento de la lista que ocupa el índice dado con la nueva cadena.
void select(int índice)	Selecciona el elemento de la lista que ocupa el índice dado.
void setMultipleMode(boolean b)	Establece el indicador que determina si la lista permite selección múltiple.
void setMultipleSelections(boolean b)	Obsoleto. Reemplazado por *setMultipleMode(boolean).*

Veamos un ejemplo. En este caso, se añadirán elementos a una lista, haciendo que sólo cuatro de ellos se vean de una vez. Cuando el usuario haga doble clic sobre uno de ellos, se notificará en un cuadro de texto que uno de

ellos se ha seleccionado. Empezaremos creando la lista y después haremos que sólo cuatro líneas sean visibles, como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
CODE=lista.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class lista extends Applet implements ActionListener {

    List list1;
    TextField text1;

    public void init(){
        text1 = new TextField(20);
        add(text1);

        list1 = new List(4);
        .
        .
        .
    }
}
```

Ahora, se añaden nueve elementos usando el método *add:*

```
public void init(){
        text1 = new TextField(20);
        add(text1);

        list1 = new List(4);
        list1.add("Elemento 1");
        list1.add("Elemento 2");
        list1.add("Elemento 3");
        list1.add("Elemento 4");
        list1.add("Elemento 5");
        list1.add("Elemento 6");
        list1.add("Elemento 7");
        list1.add("Elemento 8");
        list1.add("Elemento 9");
        add(list1);
        .
        .
        .
   }
```

Después de añadir la lista a este esquema, se le añade un *listener*. Esto significa que las listas utilizan las interfaces *ActionListener*, al igual que los botones. Así es como añadimos esto al código:

```
public void init(){
        text1 = new TextField(20);
        add(text1);

        list1 = new List(4);
        list1.add("Elemento 1");
        list1.add("Elemento 2");
        list1.add("Elemento 3");
        list1.add("Elemento 4");
        list1.add("Elemento 5");
        list1.add("Elemento 6");
        list1.add("Elemento 7");
        list1.add("Elemento 8");
        list1.add("Elemento 9");
        add(list1);

        list1.addActionListener(this);
    }
```

Ahora necesitamos añadir un método *actionPerformed* al *main* de la clase *applet* para gestionar los dobles clic (a este método no se le llama con clic simples, que es cuando se seleccionan los elementos en una lista):

```
public void actionPerformed(ActionEvent e)
{
    .
    .
    .
}
```

En este caso, se obtendrá el elemento sobre el que el usuario ha hecho doble clic usando el método *getSelectedItem* de la clase *List* (el primer clic del doble clic selecciona el elemento) y luego se visualiza ese elemento en el cuadro de texto, como sigue:

```
public void actionPerformed(ActionEvent e)
{

        if(e.getSource() == list1){
            text1.setText(((List) e.getSource()).getSelectedItem());
        }
}
```

El resultado de este código se muestra en la figura 8.4. Cuando el usuario hace doble clic en un elemento de la lista, ese elemento se visualiza en el cuadro de texto. Observe además que sólo cuatro elementos de los nueve que hay en total, son visibles en la lista, porque la lista actual es más larga que el

número de elementos visibles. El control lista ha añadido automáticamente una barra de desplazamiento vertical. Este ejemplo se llama lista.java en el CD.

Se pueden hacer más cosas con las listas, por ejemplo, se puede permitir al usuario que haga selecciones múltiples. Echaremos un vistazo a esto en el siguiente punto.

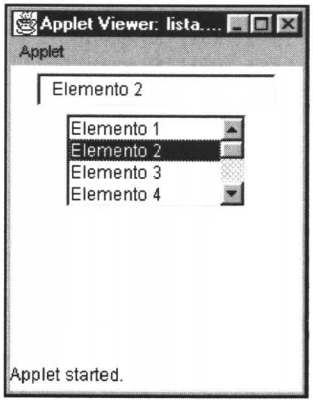


Figura 8.4. Usar una lista.

Usar listas de selección múltiple

"Uh-oh", dice el programador novato", tengo un problema". Mi nuevo programa permite a los usuarios ordenar los alimentos utilizando una lista, pero lo tienen que hacer uno por uno". "No es problema", le contesta, "basta con habilitar la selección múltiple en la lista".

Si al constructor de la clase *List* se le pasa un segundo parámetro con el valor *verdadero,* se creará una lista que soporta la selección múltiple. El usuario puede seleccionar múltiples elementos pulsando la tecla **Control** y a la vez haciendo clic o seleccionar un rango de elementos pulsando la tecla **Mayús** y haciendo clic.

Veamos un ejemplo. En este caso, permitiremos al usuario seleccionar varios elementos y cuando haga clic sobre un botón, listaremos los elementos seleccionados en un cuadro de texto. Empezaremos creando una lista de selección múltiple:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
```

```
<APPLET
    CODE=seleccionmult.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class seleccionmult extends Applet implements ActionListener {

    List list1;
    TextField text1;
    Button button1;
    String selections[];

    public void init(){
        text1 = new TextField(40);
        add(text1);
        list1 = new List(4, true);
        list1.add("Elemento 1");
        list1.add("Elemento 2");
        list1.add("Elemento 3");
        list1.add("Elemento 4");
        list1.add("Elemento 5");
        list1.add("Elemento 6");
        list1.add("Elemento 7");
        list1.add("Elemento 8");
        list1.add("Elemento 9");
        add(list1);
        button1 = new Button("Mostrar las selecciones");
        button1.addActionListener(this);
        add(button1);
    }
```

A continuación, se añade un *ActionListener* a la lista y el botón sobre el que el usuario puede hacer clic para visualizar las selecciones hechas en la lista:

```
public void init(){
        text1 = new TextField(40);
        add(text1);
        list1 = new List(4, true);
        list1.add("Elemento 1");
        list1.add("Elemento 2");
        list1.add("Elemento 3");
        list1.add("Elemento 4");
        list1.add("Elemento 5");
        list1.add("Elemento 6");
        list1.add("Elemento 7");
        list1.add("Elemento 8");
        list1.add("Elemento 9");
        add(list1);
        button1 = new Button("Mostrar las selecciones");
        button1.addActionListener(this);
        add(button1);
    }
```

Ahora, es necesario usar un botón para mostrar las selecciones múltiples que se han hecho (observe que no se permite al usuario iniciar una acción que involucre selecciones múltiples utilizando el ratón, porque el hacer clic o doble clic sobre cualquier elemento deseleccionará el resto).

Cuando el usuario hace clic sobre el botón, se usa el método *getSelectedItems* del componente *List* para obtener un *array* de tipo *String* de los elementos seleccionados:

```
String selections[];
public void actionPerformed(ActionEvent e)
{
     if(e.getSource() == button1){
             selections = list1.getSelectedItems();
              .
              .
              .
      }
}
```

A continuación, se recorre ese *array*, añadiendo todos los elementos a una cadena que comienza con "Usted seleccionó:":

```
public void actionPerformed(ActionEvent e)
    {
        String outString = new String("Usted seleccionó:");

        if(e.getSource() == button1){
            selections = list1.getSelectedItems();
               for(int loopIndex = 0; loopIndex < selections.length;
loopIndex++){
                outString += " " + selections[loopIndex];
            }
            .
            .
            .
    }
}
```

Finalmente, se sitúa la cadena con todas las selecciones en el cuadro de texto:

```
public void actionPerformed(ActionEvent e)
    {
        String outString = new String("Usted seleccionó:");

        if(e.getSource() == button1){
            selections = list1.getSelectedItems();
               for(int loopIndex = 0; loopIndex < selections.length;
loopIndex++){
                outString += " " + selections[loopIndex];
            }
```

```
            text1.setText(outString);
        }
    }
```

Y eso es todo. El resultado aparece en la figura 8.5. Como se puede ver en esta figura, el usuario puede hacer selecciones múltiples en esta nueva lista, y cuando haga clic sobre el botón, las selecciones aparecerán en el cuadro de texto. Este ejemplo se llama seleccionmult.java en el CD.

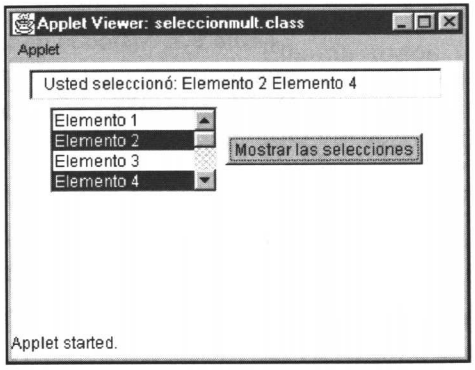


Figura 8.5. Usar la selección múltiple en una lista.

Usar cuadros de lista desplegables

"Vaya", dice el programador novato, "el gran jefe está recortando costes de nuevo, y dice que todos los controles deben tener la mitad de su tamaño original. ¡Pero el control lista que estoy utilizando es ya bastante pequeño!" "No hay problema", le dice, "basta con usar un cuadro de lista desplegable, en su lugar".

Los cuadros de lista desplegables son como las listas que sólo visualizan un elemento. Para visualizar toda la lista en un cuadro de este tipo, se hace clic sobre el botón que se muestra en la parte derecha del control. Este hace que se muestre una lista desplegable y que luego se pueda seleccionar un elemento con el ratón. Después, la lista se cierra. La selección actual aparece en el cuadro de lista desplegable cuando se cierra la lista (observe que no se puede hacer selección múltiple en este tipo de cuadros). Este es el diagrama de herencia de los cuadros de lista desplegables:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Choice
```

El constructor de la clase *Choice* se encuentra en la tabla 8.6 y sus métodos en la tabla 8.7.

Tabla 8.6. Constructor de la clase *Choice*.

Constructor	Descripción
Choice()	Crea un nuevo cuadro de lista desplegable.

Tabla 8.7. Métodos de la clase *Choice*.

Método	Descripción
void add(String elemento)	Añade un elemento a este control.
void addItem(String elemento)	Añade un elemento a este control.
void addItemListener(ItemListener l)	Añade el *item listener* dado para obtener los eventos de este control.
void addNotify()	Crear el compañero de este control.
int countItems()	Obsoleto. Reemplazado por *getItemCount().*
String getItem(int índice)	Obtiene la cadena de este control que ocupa el índice dado.
int getItemCount()	Obtiene el número de elementos de este control.
int getSelectedIndex()	Obtiene el índice del elemento seleccionado actualmente.
String getSelectedItem()	Obtiene una representación de tipo *string* de la selección actual.
Object[] getSelectedObjects()	Obtiene un *array* (longitud 1) que contiene el elemento seleccionado actualmente.
void insert(String elemento, int *índice)*	Inserta el elemento en la posición dada.
protected String paramString()	Obtiene la representación tipo *string* del estado de este control.
protected void processEvent (AWTEvent e)	Procesa los eventos en este control.
protected void processItemEvent (ItemEvent e)	Procesa los eventos de elemento que ocurren en este control, envián-

Método	Descripción
	dolos a cualquiera de los objetos *ItemListener* registrados.
void remove(int posición)	Elimina el elemento que está en la posición indicada.
void remove(String elemento)	Elimina la primera ocurrencia de un elemento de este control.
void removeAll()	Elimina todos los elementos del control.
void removeItemListener(ItemListener l)	Elimina un *item listener* para que no obtenga más eventos de elemento de este control.
void select(int pos)	Fija el elemento seleccionado de este control a la posición indicada.
void select(String str)	Fija el elemento seleccionado de este control al elemento que tiene el mismo nombre que la cadena dada.

Veamos un ejemplo. En este caso, se añadirá un cuadro de lista desplegable a un programa y se usará el método *add* para añadir los elementos a su lista interna. Cuando el usuario haga una selección, se visualizará esa nueva selección en un cuadro de texto.

Empezaremos creando una nueva lista desplegable y su lista interna (observe que no se tienen muchas opciones para crear este tipo de controles, sólo hay un constructor, y no tiene parámetros):

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=choice.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class choice extends Applet implements ItemListener {

  TextField text1;
  Choice choice1;

  public void init(){
```

```
        text1 = new TextField(20);
        add(text1);

        choice1 = new Choice();
        choice1.add("Elemento 1");
        choice1.add("Elemento 2");
choice1.add("Elemento 3");
choice1.add("Elemento 4");
choice1.add("Elemento 5");
choice1.add("Elemento 6");
choice1.add("Elemento 7");
choice1.add("Elemento 8");
choice1.add("Elemento 9");
choice1.add("Elemento 10");
choice1.add("Elemento 11");
choice1.add("Elemento 12");
   }
   .
   .
   .
}
```

Con un gran número de elementos como ocurre aquí, el cuadro de lista desplegable añadirá automáticamente una barra de desplazamiento vertical.

Ahora, gestionaremos los eventos del cuadro de lista desplegable. Con este tipo de controles, usaremos las interfaces *ItemListener*, no las *ActionListener* como se hace con los controles de tipo lista. La interfaz *ItemListener* sólo tiene un método, *itemStateChanged*:

```
void itemStateChanged(ItemEvent e)
```

A este método se le pasa un objeto de la clase *ItemEvent*. Los campos de esta clase se muestran en la tabla 8.8, su constructor en la tabla 8.9 y sus métodos en la 8.10.

Tabla 8.8. Campos de la clase *ItemEvent*.

Campo	Descripción
static int DESELECTED	Indica que un elemento seleccionado ha sido deseleccionado.
static int ITEM_FIRST	El primer número del rango de ID's usado para los eventos de elemento.
static int ITEM_LAST	El último número del rango de ID's usado para los eventos de elemento.
static int ITEM_STATE_CHANGED	Indica que el estado de un elemento cambió.
static int SELECTED	Indica que un elemento estaba seleccionado.

Tabla 8.9. Constructor de la clase *ItemEvent*.

Constructor	Descripción
ItemEvent(ItemSelectable source, int id, Object item, int stateChange)	Construye un objeto *ItemEvent*.

Tabla 8.10. Métodos de la clase *ItemEvent*.

Método	Descripción
Object getItem()	Obtiene el elemento afectado por el evento.
ItemSelectable getItemSelectable()	Obtiene el que origina el evento.
int getStateChange()	Obtiene el tipo de cambio de estado (es decir, seleccionado o deseleccionado).
String paramString()	Obtiene una cadena que identifica este evento.

A continuación se añade un *ItemListener* al cuadro de lista desplegable, como sigue:

```
public void init()
{
   text1 = new TextField(20);
   add(text1);

   choice1 = new Choice();
   choice1.add("Elemento 1");
   choice1.add("Elemento 2");
choice1.add("Elemento 3");
choice1.add("Elemento 4");
choice1.add("Elemento 5");
choice1.add("Elemento 6");
choice1.add("Elemento 7");
choice1.add("Elemento 8");
choice1.add("Elemento 9");
choice1.add("Elemento 10");
choice1.add("Elemento 11");
choice1.add("Elemento 12");
   add(choice1);

   choice1.addItemListener(this);
}
```

Cuando el usuario hace una selección de la lista interna, usaremos el método *getSelectedItem* del cuadro de lista desplegable para determinar qué elemento se seleccionó y luego lo visualizaremos en un cuadro de texto:

```
public void itemStateChanged(ItemEvent e)
{
   if(e.getItemSelectable() == choice1){
        text1.setText("Usted eligió " +
              ((Choice)e.getItemSelectable()).getSelectedItem());
   }
}
```

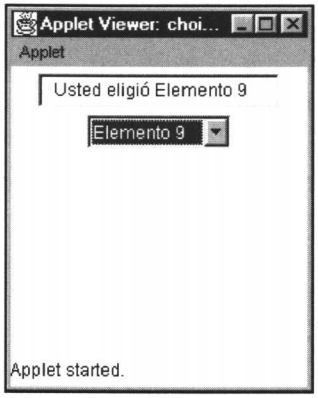


Figura 8.6. Usar cuadros de lista desplegables.

El resultado aparece en la figura 8.6. Como se puede ver en esta figura, los usuarios pueden seleccionar un elemento del cuadro de lista desplegable, y cuando lo hacen, el programa visualiza ese elemento en el cuadro de texto.

Usar barras de desplazamiento

El programador novato aparece y dice, "Tengo un problema. Quiero que los usuarios puedan seleccionar, en mi nuevo programa, el color para dibujar, pero están hartos de escribir los valores del color en formato hexadecimal". Se sonríe y le dice, "¿Por qué no usa las barras de desplazamiento? Son el componente perfecto cuando se quiere que los usuarios puedan seleccionar de un rango numérico continuo". "Suena bien", dice PN.

Todos los usuarios GUI están familiarizados con las barras de desplazamiento. En Java, las barras de desplazamiento están formadas por unas flechas de desplazamiento (los botones que hay en cada extremo de la barra de desplazamiento), un cuadro de desplazamiento (cuadro que se puede deslizar) y un área de desplazamiento (parte de la barra de desplazamiento por la que se desliza el cuadro de desplazamiento). Las barras AWT se soportan con la clase *Scrollbar*, que tiene el siguiente diagrama de herencia:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Scrollbar
```

Los constructores de la clase *Scrollbar* se encuentran en la tabla 8.11 y sus métodos en la tabla 8.12.

Las barras de desplazamiento no usan las interfaces *ActionListener* ni *ItemListener*; en su lugar, utilizan las interfaces *AdjustmentListener*. Esta interfaz sólo tiene un método, *adjustmentValueChanged:*

```
void adjustmentValueChanged(AdjustmentEvent e)
```

Tabla 8.11. Constructores de la clase **Scrollbar.**

Constructor	Descripción
Scrollbar()	Crea una nueva barra de desplazamiento vertical.
*Scrollbar(int*orientación)	Crea una nueva barra de desplazamiento con la orientación dada.
Scrollbar(int orientación, *int* valor, *int* visible, *int* mínimo, *int* máximo)	Crea una nueva barra de desplazamiento con la orientación, valor inicial, tamaño de la página y valores mínimo y máximo dados.

Tabla 8.12. Métodos de la clase *Scrollbar.*

Método	Descripción
void addAdjustmentListener(AdjustmentListener l)	Añade el *adjustment listener* dado, para obtener los eventos de ajustes.
void addNotify()	Crea el compañero de la barra de desplazamiento.
int getBlockIncrement()	Obtiene el incremento de bloque de la barra de desplazamiento.
int getLineIncrement()	Obsoleto. Reemplazado por*getUnitIncrement().*
int getMaximum()	Obtiene el valor máximo de la barra de desplazamiento.
int getMinimum()	Obtiene el valor mínimo de la barra de desplazamiento.
int getOrientation()	Obtiene la orientación de la barra de desplazamiento.

Método	Descripción
int getPageIncrement()	Obsoleto. Reemplazado por *getBlockIncrement()*.
int getUnitIncrement()	Obtiene la unidad de incremento para esta barra de desplazamiento.
int getValue()	Obtiene el valor actual de la barra de desplazamiento.
int getVisible()	Obsoleto. Reemplazado por *getVisibleAmount()*.
int getVisibleAmount()	Obtiene la cantidad visible de la barra de desplazamiento.
protected String paramString()	Obtiene una representación de tipo *string* del estado de la barra de desplazamiento.
protected void processAdjustment Event(AdjustmentEvent e)	Procesa los eventos de ajuste que tienen lugar en esta barra de desplazamiento enviándolos a cualquiera de los objetos *AdjustmentListener* registrados.
protected void processEvent (AWTEvent e)	Procesa los eventos de la barra de desplazamiento.
void removeAdjustmentlistener (AdjustmentListener l)	Elimina el *Adjustment listener* dado, para que no se obtengan más eventos de ajuste de la barra de desplazamiento.
void setBlockIncrement(int v)	Fija el incremento de bloque para la barra de desplazamiento.
void setBlockIncrement(int v)	Obsoleto. Reemplazado por *setUnitIncrement(int)*.
void setMaximum(int newMaximum)	Establece el valor máximo.
void setMinimum(int newMinimum)	Establece el valor mínimo.
void setOrientation(int orientación)	Fija la orientación.
void setPageincrement(int v)	Obsoleto. Reemplazado por *setBlockIncrement()*.
void setUnitIncrement(int v)	Fija la unidad de incremento.
void setValue(int newValue)	Fija el valor de la barra de desplazamiento.

Método	Descripción
void setValues(int valor, *int* visible, *int* mínimo, *int* máximo)	Fija cuatro valores para la barra de desplazamiento.
void setVisibleAmount(int newAmount)	Fija qué cantidad de barra de desplazamiento es visible.

A este método *adjutementValueChanged,* se le pasa un objeto de la clase *AdjustmentEvent*. Los campos de esta clase se encuentran en la tabla 8.13, su constructor en la tabla 8.14 y sus métodos en la tabla 8.15.

Observe, en particular, que se puede usar el método *getAdjustmentType* de la clase *AdjustmentEvent* para determinar el tipo de evento de barra de desplazamiento que se ha producido, según lo especificado en los campos de la tabla 8.13.

Tabla 8.13. Campos de la clase *AdjustmentEvent.*

Campo	Descripción
ADJUSTMENT_FIRST	Primer ID entero del rango de ID's de eventos de ajuste.
ADJUSTMENT_LAST	Marca el último ID entero del rango de ID's de eventos de ajuste.
ADJUSTMENT_VALUE_CHANGED	Valor de ajuste cambiado cuando se produce un evento.
TRACK	El usuario arrastró el cuadro de desplazamiento.
UNIT_INCREMENT	El usuario hizo clic sobre la flecha de desplazamiento hacia la izquierda o sobre la de desplazamiento hacia arriba (o ejecutó la acción equivalente con el teclado).
UNIT_DECREMENT	El usuario hizo clic sobre la flecha de desplazamiento hacia la derecha o sobre la de desplazamiento hacia abajo (o ejecutó la acción equivalente con el teclado).
BLOCK_INCREMENT	El usuario hizo clic sobre la barra de desplazamiento, hacia la izquierda del cuadro de desplazamiento en una barra de desplazamiento horizontal,

Campo	Descripción
	o por encima del cuadro en una barra de desplazamiento vertical. La tecla Repág es equivalente.
BLOCK_DECREMENT	El usuario hizo clic sobre la barra de desplazamiento, hacia la derecha del cuadro de desplazamiento en una barra de desplazamiento horizontal, o por encima del cuadro en una barra de desplazamiento vertical. La tecla Avpág es equivalente.

Tabla 8.14. Constructor de la clase *AdjustmentEvent*.

Constructor	Descripción
AdjustmentEvent(Adjustable, source, int type, int value)	Construye un objeto *AdjustmentEvent*, con el id ajustable fuente, tipo de evento, tipo de ajuste y valor dados.

Tabla 8.15. Métodos de la clase *AdjustmentEvent*.

Método	Descripción
Adjustable getAdjustable()	Obtiene el objeto ajustable en el que se originó el evento.
int getAdjustmentType()	Obtiene el tipo de ajuste que produjo el evento que cambió el valor.
int getValue()	Obtiene el valor actual del evento de ajuste.
String paramString()	Obtiene una representación tipo*string* del estado del evento.

Veamos un ejemplo. Aquí, añadiremos dos barras de desplazamiento a un programa y visualizaremos sus propiedades cuando el usuario se desplaza por ellas. Cuando se construye una barra de desplazamiento, se puede especificar su orientación (horizontal o vertical), su valor inicial, tamaño de página y su rango numérico. El tamaño de página indica el tamaño del cuadro de desplazamiento (se tiene por costumbre utilizar el tamaño del cuadro de desplazamiento para indicar el rango total, se usa un cuadro más pequeño para un rango más grande y uno más grande para un rango más pequeño, como se puede ver en los procesadores de texto cuando se trabaja con documentos de

distintos tamaños). Así es cómo se añade una barra de desplazamiento a un programa, dándole una orientación horizontal, un valor inicial de 1, un tamaño de página de 20 y un rango de 1 a 200:

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;

/*
<APPLET
    CODE=desplazamiento.class
    WIDTH=400
    HEIGHT=100 >
</APPLET>
*/

public class desplazamiento extends Applet implements AdjustmentListener
{

    TextField text1;
    Scrollbar scroll1, scroll2;

    public void init()
    {
        scroll1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 20, 1, 200);
        add(scroll1);
        scroll1.addAdjustmentListener(this);
        .
        .
        .
```

Observe que además hemos añadido un *AdjustmentListener* a esta nueva barra de desplazamiento.



Truco: se podría pensar que esta barra de desplazamiento puede devolver cualquier valor dentro del rango de 1 a 200, pero de hecho, el cuadro de desplazamiento debe además estar representado en el cuadro de desplazamiento, por lo que la barra sólo puede devolver valores de 1 a 200 menos el tamaño de la página (20, en este caso), que es igual a 180. Es algo en lo que hay que pensar cuando se crean barras de desplazamiento.

La barra de desplazamiento también se puede configurar usando los métodos *setUnitIncrement* y *setBlockIncrement*. El método *setUnitIncrement* establece la cantidad de cambios de propiedades de la barra de desplazamiento cuando el usuario hace clic sobre una fecha de desplazamiento (el defecto es 1), y el método *setBlockIncrement* fija la cantidad de cambios de propie-

dad que tienen lugar cuando el usuario hace clic en el cuadro de desplazamiento (el defecto es 10).

A continuación, se añade un cuadro de texto que visualiza las nuevas propiedades de las barras de desplazamiento, en el caso de que sea vertical:

```
public void init()
    {
        scroll1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 20, 1, 200);
        add(scroll1);
        scroll1.addAdjustmentListener(this);
        text1 = new TextField(20);
        add(text1);
        scroll2 = new Scrollbar(Scrollbar.VERTICAL, 1, 20, 1, 200);
        add(scroll2);
        scroll2.addAdjustmentListener(this);
    }
```

Todo lo que queda es actualizar lo que se visualiza en el cuadro de texto cuando el usuario utiliza las barras de desplazamiento. Esto lo hacemos con el método *adjustmentValueChanged,* verificando el método *getAdjustable* del objeto *AdjustmentEvent* para asegurarse que se está tratando una de las barras de desplazamiento, y usando el método *getValue* de la barra de desplazamiento para visualizar las nuevas propiedades del cuadro de texto. Así sería el código:

```
public void adjustmentValueChanged(AdjustmentEvent e)
    {
        if(e.getAdjustable() == scroll1 ||
            e.getAdjustable() == scroll2) {
            text1.setText("Horizontal: " + scroll1.getValue() +
                " Vertical: " + scroll2.getValue());
        }
    }
```

El resultado se muestra en la figura 8.7. Como se puede ver en esta figura, el usuario puede mover las barras de desplazamiento y sus nuevas propiedades aparecerán en el cuadro de texto. Este ejemplo está en el CD en desplazamiento.java.

Eso está bien si se quieren usar barras de desplazamiento que permitan al usuario especificar fácilmente los números de cualquier rango continuo, pero ¿qué pasa si se quieren usar barras de desplazamiento para recorrer algo? A continuación, veremos otro ejemplo que recorre una cadena de texto en un a*pplet*.

En este caso, moveremos la cadena "¡Hola desde Java!" con un a*pplet* para obtener las propiedades de una barra de desplazamiento. Empezaremos creando la barra de desplazamiento y pintando la cadena en la coordenada (x, 60) donde ajustaremos x cuando se mueva la barra de desplazamiento:

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;

/*
<APPLET
    CODE=desplazamiento2.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class desplazamiento2 extends Applet implements AdjustmentListener
{
```

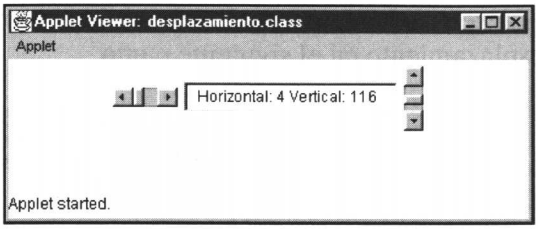


Figura 8.7. Usar barras de desplazamiento.

```
Scrollbar scroll1, scroll2;
    int x = 0;

    public void init()
    {
        scroll1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 10, 1, 100);
        add(scroll1);
        scroll1.addAdjustmentListener(this);
    }

    public void paint(Graphics g)
    {
        g.drawString("¡Hola desde Java!", x, 60);

    }
    .
    .
    .
}
```

Cuando se produzcan los eventos de desplazamiento, lo notificaremos en el método *adjustmentValueChanged*, y se puede fijar el valor de la variable x para reflejar la nueva situación horizontal de la cadena y volver a dibujar la *applet* (observe que se está usando el método *getSize* de la clase *Applet* para determinar la anchura de la *applet* y escalar los movimientos de la cadena de texto de acuerdo con ello):

```
    public void adjustmentValueChanged(AdjustmentEvent e)
        {
            if(e.getAdjustable() == scroll1) {
                x = (int) (getSize().width * (float) scroll1.getValue() /
100);
                repaint();
            }
        }
    }
```

El resultado se muestra en la figura 8.8. Como se puede ver en esta figura, el usuario puede desplazarse por la cadena de texto simplemente manipulando la barra de desplazamiento horizontal. Este ejemplo se encuentra en el CD como desplazamiento2.java.

Veremos con más detalle el desplazamiento por textos con las barras de desplazamiento en el siguiente punto.



Figura 8.8. Desplazamiento por una cadena de texto.

Barras de desplazamiento y *border layouts*

"Vaya", dice el programador novato, "ahora el gran jefe quiere que añada a mi programa barras de desplazamiento horizontal y vertical, pero no se mantienen en el lugar en que yo las puse". "Eso se debe a que debería usar un *border layout*", le contesta.

Dado que los *border layouts* permiten añadir controles alrededor del perímetro de un programa, es natural utilizarlos con barras de desplazamiento. Este es un ejemplo en el que añadiremos cuatro barras de desplazamiento alrededor del panel central. Este panel visualiza el texto "¡Hola desde Java!" en la posición (x, y), como sigue:

```
class textPanel extends Panel
{
  TextField Text1;

  public int x = 0, y = 0;

  public void paint (Graphics g)
  {
      g.drawString("¡Hola desde Java!", x, y);
  }
}
```

A continuación, se puede añadir un objeto de esta nueva clase *panel* en el centro del *border layout,* rodeándolo con barras de desplazamiento, como sigue:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
<APPLET
    CODE=desplazamientoenborde.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class desplazamientoenborde extends Applet implements
AdjustmentListener
{

    Scrollbar hScroll1, hScroll2, vScroll1, vScroll2;
    textPanel t1;

    public void init()
    {
        setLayout(new BorderLayout());

        hScroll1 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 1, 1, 100);
        add("North", hScroll1);
        hScroll1.addAdjustmentListener(this);

        vScroll1 = new Scrollbar(Scrollbar.VERTICAL, 1, 1, 1, 100);
        add("West", vScroll1);
        vScroll1.addAdjustmentListener(this);

        hScroll2 = new Scrollbar(Scrollbar.HORIZONTAL, 1, 1, 1, 100);
        add("South", hScroll2);
        hScroll2.addAdjustmentListener(this);

        vScroll2 = new Scrollbar(Scrollbar.VERTICAL, 1, 1, 1, 100);
        add("East", vScroll2);
```

```
        vScroll2.addAdjustmentListener(this);

        t1 = new textPanel();
        add("Center", t1);
    }
    .
    .
    .
}
```

El único truco es que cuando el usuario mueve una barra de desplazamiento, hay que mover la correspondiente en el otro lado del panel central para mantener la coordinación entre las barras de desplazamiento. Así es como se haría con el método *adjustmentValueChanged:*

```
public void adjustmentValueChanged(AdjustmentEvent e)
    {
        if(e.getAdjustable() == hScroll1){
            hScroll2.setValue(hScroll1.getValue());
        }
        if(e.getAdjustable() == vScroll1){
            vScroll2.setValue(vScroll1.getValue());
        }
        if(e.getAdjustable() == hScroll2){
            hScroll1.setValue(hScroll2.getValue());
        }
        if(e.getAdjustable() == vScroll2){
            vScroll1.setValue(vScroll2.getValue());
        }
        .
        .
        .
    }
```

Lo único que queda es obtener las propiedades de las nuevas barras de desplazamiento, ajustar la posición x e y de la cadena de texto en el panel y luego volver a dibujar el panel. El proceso es el siguiente:

```
public void adjustmentValueChanged(AdjustmentEvent e)
    {
        if(e.getAdjustable() == hScroll1){
            hScroll2.setValue(hScroll1.getValue());
        }
        if(e.getAdjustable() == vScroll1){
            vScroll2.setValue(vScroll1.getValue());
        }
        if(e.getAdjustable() == hScroll2){
            hScroll1.setValue(hScroll2.getValue());
        }
        if(e.getAdjustable() == vScroll2){
            vScroll1.setValue(vScroll2.getValue());
```

```
            }

            t1.x = (int) (getSize().width * (float) hScroll1.getValue() /
100);
            t1.y = (int) (getSize().height * (float) vScroll1.getValue() /
100);
            t1.repaint();
        }
```

Y eso es todo. El resultado se muestra en la figura 8.9. Como se puede ver en ella, las barras de desplazamiento aparecen alrededor del perímetro de la *applet*. Este ejemplo está en el CD como desplazamientoenborde.java.

Hay otra forma fácil de desplazarse por el texto en paneles: se pueden usar las áreas de desplazamiento, que lo veremos después.

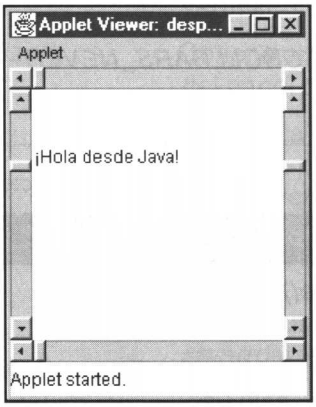


Figura 8.9. Barras de desplazamiento en un *border layout*.

Usar cuadros de desplazamiento

"Quiero desplazar algo de texto", dice el programador novato, "pero la configuración de las barras de desplazamiento es demasiado trabajo. ¿No hay una forma más fácil?" "Claro", le dice. "Puede usar los cuadros de desplazamiento". "¡Bien!", dice PN.

Se puede añadir un componente a un cuadro de desplazamiento, y le permitirá desplazarse alrededor del componente. Si el componente es más grande que el cuadro de desplazamiento, sólo será visible de una vez una parte del componente. Este es el diagrama de herencia de la clase *ScrollPane*:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Container
          |____java.awt.ScrollPane
```

Se pueden encontrar los campos de la clase *ScrollPane* en la tabla 8.16, sus constructores en la tabla 8.17 y sus métodos en la 8.18.

Tabla 8.16. Campos de la clase *ScrollPane*.

Campo	Descripción
static int SCROLLBARS_ALWAYS	Indica que las barras de desplazamiento horizontal/vertical siempre deberían aparecer.
static int SCROLLBARS_AS_NEED	Indica que las barras de desplazamiento horizontal/vertical deberían aparecer sólo cuando el tamaño del hijo sea mayor que el del panel de desplazamiento.
static int SCROLLBARS_NEVER	Indica que las barras de desplazamiento horizontal/vertical no se deberían mostrar nunca.

Tabla 8.17. Constructores de la clase *ScrollPane*.

Constructor	Descripción
ScrollPane()	Crea un nuevo cuadro de desplazamiento.
ScrollPane(int scrollbarDisplayPolicy)	Crea un nuevo cuadro de desplazamiento, utilizando una política de visualización.

Tabla 8.18. Métodos de la clase *ScrollPane*.

Método	Descripción
protected void addImpl(Component como, Object constraints, int índice)	Añade al cuadro de desplazamiento el componente dado.
void addNotify()	Crea el compañero del cuadro de desplazamiento.
void doLayout()	Adapta este contenedor redimensionando el componente contenido a su tamaño preferido.
Adjustable getHAdjustable()	Obtiene la barra de desplazamiento horizontal.

Método	Descripción
int getHScrollbarHeight()	Obtiene la altura que debería ocupar una barra de desplazamiento horizontal.
int getScrollbarDisplayPolicy()	Obtiene la política de visualización de las barras de desplazamiento.
Point getScrollPosition()	Obtiene la posición actual x, y dentro del hijo, que es visualizado en el origen de la parte visible del panel de desplazamiento.
Adjustable getVAdjustable()	Obtiene la barra de desplazamiento vertical.
Dimension getViewportSize()	Obtiene el tamaño actual de la vista del panel de desplazamiento.
int getVScrollbarWidth()	Obtiene la anchura que debería ser ocupada por la barra de desplazamiento vertical.
void layout()	Obsoleto. Reemplazado por *doLayout()*.
String paramString()	Obtiene una representación de tipo *string* del estado de este contenedor.
void printComponents(Graphics g)	Visualiza el componente en este cuadro de desplazamiento.
void setLayout(LayoutManager mgr)	Fija el *layout manager* para este contenedor.
void setScrollPosition(int x, *int* y)	Desplaza a la posición dada dentro del componente hijo.
void setScrollPosition(point p)	Desplaza a la posición dada dentro del componente hijo.

Veamos un ejemplo. Es bastante fácil añadir un componente a un cuadro de desplazamiento, simplemente hay que usar el método *add* de la clase *ScrollPane*. Así es como se haría cuando se quiere añadir un cuadro de texto a un cuadro de desplazamiento:

```
import java.applet.Applet;
import java.awt.*;

/*
```

```
<APPLET
    CODE=cuadrodedesplazamiento.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class cuadrodedesplazamiento extends Applet
{

    ScrollPane scrollpane1;
    TextPanel t1;

    public void init()
    {
   scrollpane1 = new Scrollpane();
   text1 = new TextField("¡Hola desde Java!");
   scrollpane1.add(text1);
   add(scrollpanel);
    }
}
```

Sin embargo, es curioso cuando sólo una parte del cuadro de texto es visible en un cuadro de desplazamiento. Más interesante sería incluir desplazamiento de texto en un panel, y ahora veremos un ejemplo. En este caso, crearemos una nueva clase *panel, TextPanel,* que visualiza una larga cadena de texto:

```
class TextPanel extends Panel
{

    public void paint (Graphics g)
    {
        g.drawString ("Esta es una cadena de texto larga que " +
            "parece continuar y continuar y continuar....", 0, 60);
    }
}
```

Hay un punto importante que la mayoría de los libros nunca tratan: cuando se usa un objeto que no tiene un tamaño predefinido, como un panel en un cuadro de desplazamiento, se tiene que dar un tamaño. En el caso de los contenedores y de los paneles, eso quiere decir que hay que sobrescribir el método *getPreferredSize* para que se pueda llamar al método desde el cuadro de desplazamiento. Aquí se devuelve un objeto de la clase *Dimension,* que sólo tiene dos miembros de datos, anchura y altura. Estos valores se establecen como sigue:

```
class TextPanel extends Panel
{
    public Dimension getPreferredSize()
    {
        return new Dimension(400, 400);
```

```
    }

    public void paint (Graphics g)
    {
        g.drawString ("Esta es una cadena de texto larga que " +
            "parece continuar y continuar y continuar....", 0, 60);
    }
}
```

Ahora, añadiremos un objeto de la clase *TextPane* a un cuadro de desplazamiento, como sigue:

```
import java.applet.Applet;
import java.awt.*;

/*
<APPLET
    CODE=cuadrodedesplazamiento.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class scrollpane extends Applet
{

    ScrollPane scrollpane1;
    TextPanel t1;

    public void init(){
        scrollpane1 = new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);

        t1 = new TextPanel();
        scrollpane1.add(t1);
        add(scrollpane1);
    }
}
```

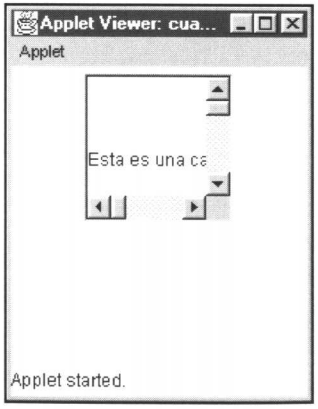


Figura 8.10. Desplazarse por un texto utilizando un cuadro de desplazamiento.

El resultado se muestra en la figura 8.10. Como se puede ver en esta figura, el usuario finaliza el desplazamiento por el texto en el interior de un cuadro de desplazamiento, en lugar de desplazarse por algún control como es un cuadro de texto. Este ejemplo se encuentra en el CD como cuadrodedesplazamiento.java.

9 AWT: Gráficos, imágenes, texto y fuentes

Este capítulo trata sobre los temas potenciales de Java: gráficos, imágenes, gestión de textos y el trabajo con las fuentes. Java es un lenguaje muy visual y todas estás áreas son conocidas por los programadores. Empezaremos con una visión rápida de estos elementos.

Gráficos

La capacidad de gráficos de AWT es bastante sólida y se basa en la gran clase *Graphics*. Dicha clase se puede utilizar para dibujar cualquier tipo de figuras, rectas, puntos, rectángulos, óvalos, polígonos y algunos otros. Además, es posible seleccionar los colores, los modos de dibujo y colorear las figuras. Veremos todo esto en este capítulo, incluyendo también un componente especial, *Canvas*, que existe expresamente para que se pueda dibujar en él.

Imágenes

En AWT hay que destacar la gestión de imágenes y veremos lo que es capaz de hacer a lo largo de este capítulo. Echaremos un vistazo a las diversas

formas de cargar imágenes de diferentes formatos como GIF y JPG, redimensionamiento de imágenes, espera hasta que las imágenes estén totalmente cargadas antes de visualizarlas, dibujar imágenes antes de visualizarlas (proceso llamado *double buffering*), y animación de imágenes. De hecho, es posible acceder a cada uno de los pixels de las imágenes, y haremos eso aquí, copiando imágenes, dándoles brillo, convirtiéndolas a una escala de grises y dándoles una apariencia de grabado.



Truco: aprenderemos todo sobre la animación de imágenes y el *double buffering* cuando veamos los multihilos, más adelante.

Texto y fuentes

Quizás le resulte sorprendente ver que se tratan el texto y las fuentes en un capítulo de gráficos, pero cuando se escribe texto directamente, sin insertarlo en un control como un cuadro de texto, se está creando un gráfico y Java lo trata como tal. En este capítulo, veremos un poco más sobre el trabajo con textos como si fueran gráficos. Por ejemplo, se verá cómo fijar la fuente y el estilo del texto, como cursiva o negrita, y cómo medir el tamaño de una cadena de texto en la pantalla, por lo que se puede centrar en una ventana.

Teclado y ratón

En este capítulo, escribiremos código que permita visualizar texto directamente, sin ningún control como cuadros de texto o áreas de texto, lo que quiere decir que tendremos que leer ese texto directamente del teclado. Por lo tanto, veremos aquí la gestión de la entrada de datos por el teclado. Adicionalmente, cuando se está permitiendo al usuario la creación de gráficos, el ratón es una herramienta útil; de hecho, en este capítulo crearemos un programa de dibujo con ratón, por lo tanto, echaremos un vistazo a cómo se codifica el uso del ratón. Empezaremos este capítulo revisando el uso del ratón y del teclado.

Eso es todo. Ya hemos revisado rápidamente lo que hay en este capítulo. Pasemos a la siguiente sección.

Usar el ratón

"De acuerdo", dice el programador novato, "en mi programa, el usuario puede seleccionar texto con **Control-Alt-F8**, mover el cursor de inserción

con **Mayús-Alt-F3**, y..." "Espere un minuto", le dice. "¿Ha pensado en incorporar el soporte del ratón a su programa? Será más fácil para el usuario". "Hmm", dice el programador, recapacitando.

Se puede trabajar con el ratón usando dos interfaces AWT, *MouseListener,* que gestionan los clics con el ratón, el apretar o soltar sus botones, así como el caso en que el ratón introduzca un componente y luego lo abandone, y *MouseMotionListener,* que gestiona los movimientos de ratón y las operaciones de arrastre. Los métodos de la interfaz *MouseListener* se recogen en la tabla 9.1 y los de *MouseMotionListener* en la tabla 9.2.



Truco: hay que tener en cuenta que, si se quiere implementar una interfaz, es necesario sobrescribir todos sus métodos, lo que puede ser algo tedioso en el caso de *MouseListener,* que tiene cinco métodos. Para facilitar las cosas, en su lugar, se pueden usar las clases adaptadoras como *MouseAdapter* y sólo sobrescribir los métodos necesarios. Para ver más sobre las clases adaptadoras, ir al capítulo 6.

Tabla 9.1. Métodos de la interfaz *MouseListener.*

Método	Descripción
void mouseClicked(MouseEvent e)	Se le llama cuando se ha hecho clic, con el ratón, en un componente.
void mouseEntered(MouseEvent e)	Se le llama cuando el ratón introduce un componente.
void mouseExited(MouseEvent e)	Se le llama cuando el ratón sale de un componente.
void mousePressed(MouseEvent e)	Se le llama cuando se presiona un botón del ratón en un componente.
void mouseReleased(MouseEvent e)	Se le llama cuando se suelta un botón del ratón en un componente.

Tabla 9.2. Métodos de la interfaz *MouseMotionListener.*

Método	Descripción
void mouseDragged(MouseEvent e)	Se le llama cuando se aprieta un botón del ratón en un componente y luego se arrastra.
void mouseMoved(MouseEvent e)	Se le llama cuando el ratón se ha movido sobre un componente (sin apretar botón).

A cada uno de los métodos de las interfaces de ratón se le pasa un objeto de la clase *MouseEvent*, y el diagrama de herencia de esa clase es como sigue:

```
java.lang.Object
|____java.util.EventObject
     |____java.awt.AWTEvent
          |____java.awt.event.ComponentEvent
                    |____java.awt.event.InputEvent
                              |____java.awt.event.MouseEvent
```

La clase *MouseEvent* añade sólo los campos a sus clases base. Los campos de esta clase se encuentran en la tabla 9.3.

Tabla 9.3. Campos de la clase *MouseEvent*.

Campo	Descripción
static int MOUSE_CLICKED	Indica el evento "*mouse clicked*".
static int MOUSE_DRAGGED	Indica el evento "*mouse dragged*".
static int MOUSE_ENTERED	Indica el evento "*mouse entered*".
static int MOUSE_EXITED	Indica el evento "*mouse exited*".
static int MOUSE_FIRST	Indica el primer número del rango de ID's usado para los eventos de ratón.
static int MOUSE_LAST	Indica el último número del rango de ID's usado para los eventos de ratón.
static int MOUSE_MOVED	Indica el evento "*mouse moved*"
static int MOUSE_PRESSED	Indica el evento "*mouse pressed*".
static int MOUSE_RELEASED	Indica el evento "*mouse released*".

Veamos un ejemplo. Esta *applet, mouse.java,* visualizará la mayor parte de lo que se puede hacer con el ratón. Para capturar las acciones específicas del ratón, sobrescribimos el correspondiente método *mouse listener*. Para obtener la posición actual del ratón de un objeto *MouseEvent,* se usan los métodos *getX* y *getY*. Para saber qué botón se presionó, se puede usar el método *getModifiers* de la clase *MouseEvent* y luego se añade el resultado con los campos siguientes de la clase *InputEvent:ALT_GRAPH_MASK, ALT_MASK, BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK, CTRL_MASK, META_MASK* y *SHIFT_MASK*. Así es el código de esta *applet, mouse.java:*

```
import java.applet.Applet;
import java.awt.*;
```

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

/*
<APPLET
    CODE=raton.class
    WIDTH=300
    HEIGHT=200 >
<APPLET
*/

public class raton extends Applet implements MouseListener,
MouseMotionListener
{
    TextField text1;

    public void init(){
        text1 = new TextField(35);
        add(text1);
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mousePressed(MouseEvent e)
    {
if((e.getModifiers() & InputEvent.BUTTON1_MASK) ==
InputEvent.BUTTON1_MASK){
text1.setText("Botón izquierdo del ratón apretado en " + e.getX() +
"", + e.getY());
        }
        else{
text1.setText("Botón derecho del ratón pulsado en " + e.getX() + "",
+ e.getY());
        }
    }

    public void mouseClicked(MouseEvent e)
    {
        text1.setText("Hizo clic sobre el ratón en " + e.getX() + "",
+ e.getY());
    }

    public void mouseReleased(MouseEvent e)
    {
        text1.setText("Se soltó el botón del ratón.");
    }

    public void mouseEntered(MouseEvent e)
    {
        text1.setText("Ratón para introducir.");
    }

    public void mouseExited(MouseEvent e)
    {
        text1.setText("Ratón para salir.");
    }
```

```
    public void mouseDragged(MouseEvent e)
    {
        text1.setText("Se arrastró el ratón.");
    }

    public void mouseMoved(MouseEvent e)
    {
        text1.setText("Se movió el ratón.");
    }
}
```

Se puede ver esta *applet* funcionando en la figura 9.1. Cuando se mueve el ratón o se usa uno de sus botones, la *applet* permite saber lo que ocurre. Este ejemplo está en el CD que acompaña a este libro, como raton.java.

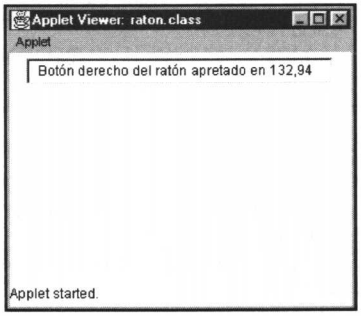


Figura 9.1. Usar el ratón.

Usar el teclado

"Bien", dice el programador novato, "estoy escribiendo un procesador de texto, *SuperDuperTextPro,* en Java y quiero leer el texto directamente del teclado. ¿Cómo funciona eso?" "Con el *key listener*", le dice. "Sabía que habría algo que me gustaría", dice PN.

Se usa la interfaz *keyListener* para trabajar con el teclado, y se pueden encontrar sus métodos en la tabla 9.4.



Truco: si se quiere implementar una interfaz, hay que tener en cuenta que es necesario sobrescribir todos sus métodos, lo que puede ser un poco tedioso. Para facilitar las cosas, se pueden usar, en su lugar, las clases adaptadoras como *KeyAdapter* y sólo se sobrescribirán los métodos que se quieran usar. Para más detalles sobre las clases adaptadoras, ver el capítulo 6.

Observe que hay tres eventos de tecla diferentes: tecla pulsada, soltar la tecla y escribir. Generalmente se usa el evento de escritura cuando se está trabajando con el teclado, porque se puede usar el método *getKeyChar* en *keyTyped* para obtener el carácter Unicode que se escribió. Por otro lado, en los métodos *KeyPressed* y *KeyReleased* se puede usar el método *getKeyCode* (no *getKeyChar)* para obtener un código virtual de la tecla; este código sólo dice que la tecla fue presionada o soltada, cada uno es responsable de averiguar si se pulsaron la tecla **Mayús**, **Control** u otra tecla, lo que se puede hacer con el objeto *KeyEvent* que se le pasa a los métodos de eventos de teclas. Para saber las teclas modificadoras (como **Mayús**) que se pulsaron, se puede usar el método *getModifiers* del objeto *KeyEvent* y luego añadir el resultado con los campos de la clase *InputEvent*: *ALT_GRAPH_MASK, ALT_MASK, CTRL_MASK, META_MASK* y *SHIFT_MASK.*

En general, no es muy fácil trabajar con los códigos virtuales porque hay una constante separada de los códigos que devuelve *getKeyCode* para cada tecla, poro ejemplo, VK_F1 para la tecla F1, VK_A para el carácter A. VK_5 para el número 5, etc., como se enumera en los campos de la clase *KeyEvent,* que se pueden ver en la tabla 9.5. Además, los constructores de esta clase se recogen en la tabla 9.6 y sus métodos en la tabla 9.7. Este es el diagrama de herencia para esta clase:

```
java.lang.Object
|____java.util.EventObject
     |____java.awt.AWTEvent
          |____java.awt.event.ComponentEvent
                    |____java.awt.event.InputEvent
                              |____java.awt.event.KeyEvent
```

Tabla 9.4. Métodos de la interfaz *KeyListener.*

Método	Descripción
void keyPressed(KeyEvent e)	Se le llama cuando se pulsa una tecla.
void keyReleased(keyEvent e)	Se le llama cuando se suelta una tecla.
void keyTyped(keyEvent e)	Se le llama cuando se escribe con una tecla.

Tabla 9.5. Campos de la clase *KeyEvent.*

CHAR_UNDEFINED	KEY_FIRST	KEY_LAST
KEY_PRESSED	KEY_RELEASED	KEY_TYPED

VK_0 a *VK_9*	*VK_A* a	*VK_Z*
VK_ACCEPT	VK_ADD	VK_AGAIN
VK_ALL_CANDIDATES	VK_ALPHANUMERIC	VK_ALT
VK_ALT_GRAPH	VK_AMPERSAND	VK_ALT
VK_AT	VK_BACK_QUOTE	VK_BACK_SLASH
VK_BACK_SPACE	VK_BRACELEFT	VK_BRACERIGHT
VK_CANCEL	VK_CAPS_LOCK	VK_CIRCUMFLEX
VK_CLEAR	VK_CLOSE_BRACKET	VK_CODE_INPUT
VK_COLON	VK_COMMA	VK_COMPOSE
VK_CONTROL	VK_CONVERT	VK_COPY
VK_CUT	VK_DEAD_ABOVEDOT VK_DEAD_ABOVERING	
VK_DEAD_ACUTE	VK_DEAD_BREVE	VK_DEAD_CARON
VK_DEAD_CEDILLA	VK_DEAD_CIRCUM-FLEX	VK_DEAD_DIERE-SIS
VK_DEAD_DOUBLE-ACUTE	VK_DEAD_GRAVE	VK_DEAD_IOTA
VK_DEAD_MACRON	VK_DEAD_OGONEK	VK_DEAD_SEMIVOI-CED_SOUND
VK_DEAD_TITLE	VK_DEAD_VOICED_ SOUND	VK_DECIMAL
VK_DELETE	VK_DIVIDE	VK_DOLLAR
VK_DOWN	VK_END	VK_ENTER
VK_EQUALS	VK_ESCAPE	VK_EURO_SIGN
VK_EXCLAMATION_ MARK	*VK_F1* a *VK_F24*	*VK_FINAL*
VK_FIND	VK_FULL_WIDTH	VK_GREATER
VK_HALF_WIDTH	VK_HELP	VK_HIRAGANA
VK_HOME	VK_INSERT	VK_INVERTED_EX-CLAMATION_MARK
Vk_JAPANESE_HIRA-GANA	VK_JAPANESE_KA-TAKANA	VK_JAPANESE_RO-MAN
VK_KANA	VK_KANJI	VK_KATAKANA

VK_KP_DOWN	VK_KP_LEFT	VK_KP_RIGHT
VK_KP_UP	VK_LEFT	VK_LEFT_PARENTHESIS
VK_LESS	VK_META	VK_MINUS
VK_MODECHANGE	VK_MULTIPLY	VK_NONCONVERT
VK_NUM_LOCK	*VK_NUMBER_SIGN*	*VK_NUMPAD0* a *VK_NUMPAD9*
VK_OPEN_BRACKET	VK_PAGE_DOWN	VK_PAGE_UP
VK_PASTE	VK_PAUSE	VK_PERIOD
VK_PLUS	VK_PREVIOUS_CANDIDATE	VK_PRINTSCREEN
VK_PROPS	VK_QUOTE	VK_QUOTEDBL
VK_RIGHT	VK_RIGHT_PARENTHESIS	VK_ROMAN_CHARACTERS
VK_SCROLL_LOCK	VK_SEMICOLON	VK_SEPARATER
VK_SHIFT	VK_SLASH	VK_SPACE
VK_STOP	VK_SUBTRACT	VK_TAB
VK_UNDEFINED	VK_UNDERSCORE	VK_UNDO
VK_UP		

Tabla 9.6. Constructores de la clase *KeyEvent*.

Constructor	Descripción
KeyEvent(Component fuente, *int id, long* cuándo, *int* modificadores, *int keyCode)*	Crea un objeto *KeyEvent*.
KeyEvent(Component fuente, *int id, long* cuándo, *int* modificadores, *int keyCode, char KeyChar)*	Crea un objeto *KeyEvent*.

Tabla 9.7. Métodos de la clase *KeyEvent*.

Método	Descripción
char getKeyChar()	Obtiene el carácter asociado con la tecla en este evento.
int getKeyCode()	Obtiene el código entero de la tecla asociado a este evento.

Método	Descripción
static String getKeyModifiersText(int boolean isActionKey()	Obtiene un *String* que describe el modificador) *keycode* , como "Inicio", "F1" o "A".
	Determina si la tecla de este evento es una tecla de acción, como se define en *Event.java.*
String paramString()	Obtiene una cadena identificando este evento.
void setKeyChar(char keyChar)	Fija el valor *KeyChar* para indicar un carácter.
void setKeyCode(int keyCode)	Fija el valor de *keyCode* para indicar una tecla física.
void setModifiers(int modificadores)	Fija los modificadores para indicar la teclas adicionales que fueron mantenidas (Mayús, Control, Alt, etc.). Definido como parte de *InputEvent.*

Veamos un ejemplo. Esta *applet* sencilla lee y visualiza las teclas, usando el evento *keyTyped* y el método *getKeyChar* para leer cada tecla. Cada vez que el usuario pulsa una tecla para escribir, el carácter correspondiente se añade al final de la cadena visualizada. Así es la *applet:*

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/*
  <APPLET
      CODE=tecla.class
      WIDTH=300
      HEIGHT=200 >
  </APPLET>
*/

public class tecla extends Applet implements KeyListener {

    String text = "";

    public void init()
    {
        addKeyListener(this);
        requestFocus();
```

```
    }

    public void paint(Graphics g)
    {
        g.drawString(text, 0, 100);
    }

    public void keyTyped(KeyEvent e)
    {
        text = text + e.getKeyChar();
        repaint();
    }

    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
}
```

Aquí hay que observar una cosa: para que la *applet,* en sí misma, reciba las pulsaciones, hay que darle el foco. (En un GUI, un elemento que tiene el foco es el que recibe las pulsaciones). Hacemos esto con el método *requestFocus* de la *applet*; si no se da el foco a la *applet* de forma explícita, no se verá ninguno de los caracteres escritos.

Los resultados de este código se muestran en la figura 9.2. Como se puede ver, el usuario puede escribir texto directamente en esta *applet,* que funciona como se esperaba. Este ejemplo está en el CD con el nombre tecla.java.

Esto es sólo el comienzo del trabajo con texto. Ahora veamos el siguiente tema, en el que empezaremos con las fuentes.

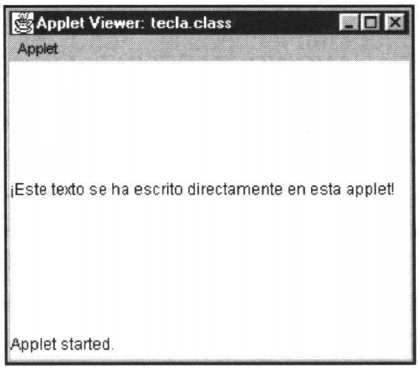


Figura 9.2. Usar el teclado.

Usar fuentes

"El *banner* que se creó para la empresa *Pride Picnic* era bueno", dice el gran jefe (GJ), "pero no parece estar hecho con orgullo". "¿Por qué no?", le

pregunta. "Por una cosa", dice GJ, "medía sólo un cuarto de pulgada de alto". "Hmm", le dice, "me quiere decir que sería mejor usar una fuente más grande".

Se puede seleccionar el tipo y el estilo de las fuentes del texto con la clase *Font*. Usando esta clase, se puede seleccionar una fuente (como helvética, arial o courier), fijar su tamaño y especificar si está en negrita, cursiva, etc. Encontrará los campos de la clase *Font* en la tabla 9.8, sus constructores en la tabla 9.9 y sus métodos en la tabla 9.10.

Por si todos los métodos de la tabla 9.10 no son suficientes, hay otra clase importante, la clase *FontMetrics*, que trata de las dimensiones físicas de las fuentes. El campo de esta clase aparece en la tabla 9.11, su constructor en la tabla 9.12 y sus métodos en la 9.13. Uno de los usos más comunes de la clase *FontMetrics* es determinar la altura del texto cuando se visualizan varias líneas de texto.

Revisemos las tablas para ver aquello de lo que disponemos cuando trabajamos con fuentes.

Tabla 9.8. Campos de la clase *Font*.

Campo	Descripción
protected Font fuente	Fuente desde la que se crean las métricas.
static int BOLD	Estilo negrita.
static int CENTER_BASELINE	Línea de base usada en los *scripts* ideográficos (como la japonesa).
static int HANGING_BASELINE	Línea de base usada cuando se esquematizan los *scripts* como *Devenagari.*
static int ITALIC	Estilo cursiva.
protected String nombre	Nombre lógico de esta fuente.
static int PLAIN	Estilo liso.
protected float pointSize	Tamaño del punto de esta fuente en un *float.*
static int ROMAN_BASELINE	Línea base usada en la mayor parte de los *scripts roman* cuando se esquematiza el texto.
protected int tamaño	Medida del punto de esta fuente.
protected int estilo	Estilo de esta fuente.

Tabla 9.9. Constructores de la clase *Font*

Constructor	Descripción
Font(Map atributos)	Crea una nueva fuente con los atributos dados.
Font(String nombre, *int* estilo, *int* tamaño)	Crea una nueva fuente con el nombre, estilo y tamaño de punto dados.

Tabla 9.10. Métodos de la clase *Font.*

Método	Descripción
boolean canDisplay(char c)	Verifica si la fuente tiene algún *glyph* para el carácter dado.
int canDisplayUpTo(char[] texto, *int* inicio, *int* límite)	Indica si esta fuente puede visualizar los caracteres del texto dado.
int canDisplayUpTo(Character-Iterator iter, int inicio, int *límite)*	Indica si esta fuente puede visualizar una cadena.
int canDisplayUpTo(String str)	Indica si esta fuente puede visualizar una cadena dada.
GlyphVector createGlypVector (Font RenderContext frc, char[] chars)	Obtiene un nuevo objeto *GlyphVector.*
GlyphVector createGlypVector (Font RenderContext frc, CharacterIterator ci)	Obtiene un nuevo objeto *GlyphVector* con el carácter iterador dado.
GlyphVector createGlypVector (Font RenderContext frc, int[] glyphCodes)	Obtiene un nuevo objeto *GlyphVector* con el *array* de enteros y *FontRenderContext* dados.
GlyphVector createGlyphVector (Font RenderContext frc, String str)	Obtiene un nuevo objeto *GlyphVector* creado con el *FontRenderContext* dado.
static Font decode(String str)	Obtiene la fuente que describe la cadena.
Font deriveFont(AffineTransform trans)	Crea un nuevo objeto *Font,* duplicando el objeto *Font* actual y aplicando una nueva transformación que convierte un conjunto de coordenadas en otras.
Font deriveFont(float tamaño)	Crea un nuevo objeto *Font,* duplicando el objeto *Font* actual y aplicando un nuevo tamaño.

Método	Descripción
Font deriveFont(int estilo)	Crea un nuevo objeto *Font* duplicando el objeto *Font* actual y aplicando el nuevo estilo.
Font deriveFont(int estilo, *Affine* y *Transform* trans)	Crea un nuevo objeto *Font,* duplicando el objeto *Font* actual aplicando un nuevo estilo y transformación.
Font deriveFont(int estilo, *float* tamaño)	Crea un nuevo objeto *Font,* duplicando el objeto *Font* actual y aplicando un nuevo estilo y tamaño.
Font deriveFont(Map atributos)	Crea un nuevo objeto *Font* duplicando el objeto *Font* actual y aplicando un nuevo conjunto de atributos de fuentes.
boolean equals(Object obj)	Compara este objeto *Font* con el objeto dado.
protected void finalize()	Elimina el objeto *Font* nativo.
Map getAtributes()	Obtiene un mapa de los atributos de esta fuente.
AttributedCharacterIterator. Attribute[] getAvailableAttributes()	Obtiene las teclas de todos los atributos de la fuente.
byte getBaselineFor(char c)	Obtiene la línea base para visualizar este carácter.
String getFamily()	Obtiene el nombre de la familia de esta fuente.
String getFamily(Locale l)	Obtiene el nombre de la familia de esta fuente, localizada en el *locale* dado.
static Font getFont(Map atributos)	Obtiene una fuente apropiada para los atributos.
static Font getFont(String nm)	Obtiene un objeto *Font* de la lista de propiedades del sistema.
static Font getFont(String nm, Font fuente)	Obtiene la fuente dada de la lista de propiedades del sistema.
String getFontName()	Obtiene el nombre de la fuente.
String getFontName(Locale l)	Obtiene el nombre de la fuente localizada en el *locale* dado.

Método	Descripción
float getItalicAngel()	Obtiene el ángulo para el estilo cursiva de esta fuente.
LineMetrics getLineMetrics (char[] chars, int beginIndex, int límite, *FontRenderContext frc)*	Obtiene un objeto *LineMetrics*, usando los argumentos dados.
LineMetrics getLineMetrics (CharacterIterator ci, int beginIndex, int límite, *FontRenderContext frc)*	Obtiene un objeto *LineMetrics* usando los argumentos dados.
LineMetrics getLineMetrics (String str, FontRenderContext frc)	Obtiene un objeto *LineMetrics*, creado con la cadena y el *FontRenderContext* dados.
Rectangle2D getMaxChar Bounds(FontRenderContext frc)	Obtiene los límites del carácter con los límites máximos, como se definió en el *FontRenderContext* dado.
int getMissingGlyphCode()	Obtiene el código *glyph* usado cuando esta fuente no tiene un *glyph* para un carácter Unicode dado.
String getName()	Obtiene el nombre lógico de la fuente.
int getNumGlyphs()	Obtiene el número de *glyphs* de esta fuente.
java.awt.peer.FontPeer getPeer()	Obsoleto. Ahora se supone que las fuentes son independientes de la plataforma.
String getPSName()	Obtiene el nombre *Postscript* de la fuente.
int getSize()	Obtiene el tamaño del punto de la fuente.
float getSize2D()	Obtiene el tamaño del punto de esta fuente en un valor *float.*
Rectangle2D getStringBounds (char[] chars, int beginIndex, int límite, *FontRenderContext frc)*	Obtiene las fronteras del *array* de caracteres en *FontRenderContext.*
Rectangle2D getStringBounds (CharacterIterator ci, int beginIndex, int límite, *FontRenderContext frc)*	Obtiene las fronteras de los caracteres en el iterador de carácter dado en *FontRenderContext.*

Método	Descripción
Rectangle2D getStringBounds (String str, FontRenderContext frc)	Obtiene los límites de la cadena dada en un *FontRenderContext.*
Rectangle2D getStringBounds (String str, int beginIndex, int límite, *FontRenderContext frc)*	Obtiene los límites de la cadena dada en un *FontRenderContext.*
int getStyle()	Obtiene el estilo de esta fuente.
Affine Transform getTransform()	Obtiene una copia de la transformación asociada con esta fuente.
int hashCode()	Obtiene un *hashcode* para esta fuente.
boolean hasUniformLineMetrics()	Verifica si esta fuente tiene líneas métricas uniformes.
boolean isBold()	Indica si el estilo de este objeto *Font* es negrita.
boolean isItalic()	Indica si el estilo de este objeto *Font* es cursiva.
boolean isPlain()	Indica si el estilo de este objeto *Font* es plano.
String toString()	Convierte el objeto *Font* en una representación de tipo *string.*

Tabla 9.12. Campo de la clase *FontMetrics.*

Campo	Descripción
protected Font fuente	Determina la fuente de la que se crean las métricas.

Tabla 9.13. Métodos de la clase *FontMetrics.*

Método	Descripción
int bytesWidth(byte[] datos, *int off, int len)*	Obtiene la anchura de avance total para mostrar el *array* de bytes dado en esta fuente.
int charsWidth(char[] datos, *int off, int len)*	Obtiene la anchura de avance total para mostrar el *array* de caracteres dado en esta fuente.

Método	Descripción
int charWidth(char ch)	Obtiene la anchura de avance del carácter dado en esta fuente.
int charWidth(int ch)	Obtiene la anchura de avance del carácter dado en esta fuente.
int getAscent()	Indica la fuente ascendente de la fuente.
int getDescendt()	Indica la fuente descendente de la fuente.
Font getFont()	Obtiene la fuente descrita por el objeto *FontMetrics.*
int getHeight()	Obtiene la altura estándar de una línea de texto en esta fuente.
int getLeading()	Indica el principal de la fuente.
LineMetrics getLineMetrics (char[] chars, int beginIndex, int límite, *Graphics* contexto)	Obtiene el objeto *LineMetrics* para el *array* de caracteres dado.
LineMetrics getLineMetrics (CharacterIterator ci, int begin-Index, int límite, *Graphics* contexto)	Obtiene el objeto *LineMetrics* para el iterador de caracteres dado.
LineMetrics getLineMetrics (Strings str, Graphics contexto)	Obtiene el objeto *LineMetrics* para la cadena dada.
LineMetrics getLineMetrics (String str, graphics int begin-Index, int límite, *Graphics* contexto)	Obtiene el objeto *LineMetrics* para la cadena dada en el contexto dado.
int getMaxAdvance()	Obtiene la máxima anchura de avance de cada carácter de esta fuente.
int getMaxAscent()	Indica el máximo ascendente de la fuente descrita por este objeto *FontMetrics.*
Rectangle2D getMaxChar-Bounds(Graphics contexto)	Obtiene los límites del carácter con los límites máximos en el contexto de gráficos dado.
int getMaxDecent()	Obsoleto. Reemplazado por *getMaxDescendt().*
int getMaxDescent()	Indica el máximo descendente de la fuente descrita por este objeto *FontMetrics.*

Método	Descripción
Rectangle2D getStringBounds (char chars, int beginIndex, int límite, *Graphics* contexto)	Obtiene los límites del *array* de caracteres dado en el contexto gráfico.
Rectangle2D getStringBounds (CharacterIterator ci, int beginIndex, int límite, *Graphics* contexto)	Obtiene los límites de los caracteres indexados en el iterador del carácter dado del contexto gráfico.
Rectangle2D getStringBounds (String str, Graphics contexto)	Obtiene los límites de la cadena dada en el contexto gráfico.
Rectangle2D getStringBounds (String str, int beginIndex, int límite, *Graphics* contexto)	Obtiene los límites de la cadena dada en el contexto gráfico.
int[] getWidths()	Obtiene la anchura de avance de los primeros 256 caracteres de la fuente.
boolean hasUniformLineMetrics()	Verifica si la fuente tiene líneas de métrica uniformes.
int stringWidth(String str)	Obtiene la anchura de avance total para mostrar la cadena dada de esta fuente.
String toString()	Obtiene los valores de este objeto *FontMetrics* como una cadena.

Veamos un ejemplo. En este caso, se permitirá al usuario escribir caracteres y visualizarlos en la fuente *Courier,* concentrado todo en un *applet,* determinando el tamaño de la pantalla del texto, usando los métodos *stringWidth* y *getHeight* de la clase *FontMetrics* y la anchura y altura de la *applet* con el método *getSize* de la *applet.* Además se permitirá al usuario especificar el tamaño del texto, así como el estilo, cursiva o negrita, y establecer un objeto *Font* acorde con ello. Para instalar la fuente, de forma que se pueda utilizar cuando se visualice texto, se usa el método *setFont* del objeto *Graphics.* Esta es la *applet:*

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/*
  <APPLET
      CODE=fuentes.class
      WIDTH=600
      HEIGHT=200 >
  </APPLET>
```

```
*/

public class fuentes extends Applet implements ActionListener, KeyListener
{

    String text = "";

    Button boldbutton, italicbutton, largebutton;
    boolean bold = false;
    boolean italic = false;
    boolean large = false;

    public void init()
    {
        boldbutton = new Button("Fuente en negrita");
        italicbutton = new Button("Fuente en cursiva");
        largebutton = new Button ("Fuente grande");

        boldbutton.addActionListener(this);
        italicbutton.addActionListener(this);
        largebutton.addActionListener(this);

        add(boldbutton);
        add(italicbutton);
        add(largebutton);

        addKeyListener(this);
        requestFocus();
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == boldbutton) bold = !bold;
        if(event.getSource() == italicbutton) italic = !italic;
        if(event.getSource() == largebutton) large = !large;
        requestFocus();
        repaint();
    }

    public void paint(Graphics g)
    {
        String fontname = "Courier";
        int type = Font.PLAIN;
        int size = 36;
        Font font;
        FontMetrics fm;

        if(bold) type = type | Font.BOLD;
        if(italic) type = type | Font.ITALIC;
        if(large) size = 72;

        font = new Font(fontname, type, size);
        g.setFont(font);

        fm = getFontMetrics(font);
```

```
            int xloc = (getSize().width - fm.stringWidth(text)) / 2;
            int yloc = (getSize().height + fm.getHeight()) / 2;

            g.drawString(text, xloc, yloc);
        }

        public void keyTyped(KeyEvent e)
        {
            text = text + e.getKeyChar();
            repaint();
        }

        public void keyPressed(KeyEvent e) {}
        public void keyReleased(KeyEvent e) {}
    }
```

El resultado se muestra en la figura 9.3. Cuando el usuario escribe texto, ese texto aparece centrado en la *applet*. Además, cuando se usan los botones fuente en negrita, fuente en cursiva y fuente grande, el texto aparece con los atributos correspondientes, que también se muestran en la figura 9.3.

Ahora es el momento de ponerse a trabajar con imágenes, veámoslo en el siguiente punto.

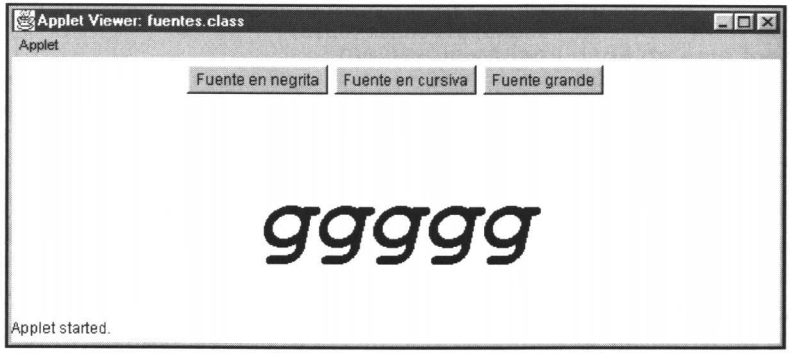


Figura 9.3. Usar fuentes.

Usar imágenes

El gran jefe dice, "En esta tentativa de reportaje que escribió para el periódico de la empresa..." "¿Sí?", le pregunta. "¿Dónde están las fotografías?", pregunta GJ. "Hmm", le dice, "parece que es un trabajo para la clase *Image*".

En AWT, el soporte de las imágenes se hace con la clase *Image*, que se deriva directamente de *java.lang.Object:*

```
java.lang.Object
|____java.awt.Image
```

Los campos de la clase *Image* se encuentran en la tabla 9.14, su constructor en la tabla 9.15 y sus métodos en la tabla 9.16.

Tabla 9.14. Campos de la clase *Image*.

Campo	Descripción
static int SCALE_AREA_AVERAGING	Indica que se utiliza el algoritmo del área media para la escala de la imagen.
static int SCALE_DEFAULT	Indica que se utiliza el algoritmo por defecto para la escala de la imagen.
static int SCALE_FAST	Indica que se ha elegido un algoritmo para la escala de imagen que da mayor prioridad a la velocidad de escalamiento.
static int SCALE_REPLICATE	Indica que se usa el algoritmo de la clase *ReplicateScaleFilter* para la escala de imagen.
static int SCALE_SMOOTH	Indica que se ha elegido un algoritmo para la escala de imagen que da más prioridad a la suavidad de la imagen.
static Object UndefinedProperty	En el caso de que no se definiera una propiedad para una imagen particular solicitada, se debería devolver el objeto *UndefinedProperty*.

Tabla 9.15. Constructor de la clase *Image*.

Constructor	Descripción
Image()	Crea un objeto *Image*.

Tabla 9.16. Métodos de la clase *Image*.

Método	Descripción
abstract void flush()	Vacía todos los recursos utilizados por la imagen.

Método	Descripción
abstract Graphics getGraphics()	Crea un contexto gráfico para dibujar una imagen por detrás de la pantalla.
abstract int getHeight(ImageObserver observer)	Indica la altura de la imagen.
abstract Object getProperty(String nombre, *ImageObserver observer)*	Obtiene una propiedad de esta imagen por nombre.
Image getScaledInstance(int anchura, *int* altura, *int* indicaciones)	Crea una versión a escala de esta imagen.
abstract ImageProducer getSource()	Obtiene el objeto que produce los pixels de la imagen.
abstract int getWidth(ImageObserver observer)	Indica la anchura de la imagen.

Para cargar una imagen en un *applet*, se puede usar el método *getImage* de la clase *Applet*:

```
Image getImage(URL url)
Image getImage(URL url, String nombre)
```

Aquí, se puede especificar la URL del fichero de imagen que se quiera leer, usando la clase URL. Se puede crear un objeto URL usando el constructor de la clase URL como sigue: *URL(http://java.sun.com/product/jdk/1.2)*, como se verá más tarde en este libro. En este capítulo, usaremos los métodos *getCodeBase* y *getDocumentBase* de la clase *Applet* para obtener la URL de la *applet*, y utilizar la misma dirección para buscar el fichero de imagen.

Este es un sencillo ejemplo que lee una imagen, *image.jpg*, y luego la visualiza. Para leer la imagen, se usa el método *getImage*. Para dibujarla, utilizamos el método *drawImage* de la clase *Graphics*. Para saber más de esta clase, ver la sección "Dibujar gráficos", más adelante en este capítulo. A continuación se indica la forma general que utilizaremos para *drawImage*, que nos permite especificar el objeto imagen que se va a dibujar y su posición en la *applet:*

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
```

Observe que a *drawImage* hay que pasarle un objeto que implemente la interfaz *ImageObserver*. Los objetos *ImageObserver* permiten monitorizar el progreso de las operaciones de carga de imágenes, y las veremos más tarde en este capítulo. En la clase *Applet*, hay una implementación por defecto de esta

interfaz y por ello, sólo utilizamos la palabra clave *this* como objeto *ImageObserver:*

```
import java.awt.*;
import java.applet.*;

/*
  <APPLET
      CODE=imagen.class
      WIDTH=500
      HEIGHT=150 >
  </APPLET>
*/

public class imagen extends Applet
{
    Image image;

    public void init()
    {
        image = getImage(getDocumentBase(), "image.jpg");
    }

    public void paint(Graphics g)
    {
        g.drawImage(image, 10, 10, this);
    }
}
```

El resultado se muestra en la figura 9.4, donde se puede ver la imagen cargada. Este ejemplo lo encontrará en el CD como imagen.java.

Por supuesto, hay muchas más cosas que se pueden hacer con las imágenes. Por ejemplo, se pueden redimensionar (ver el siguiente punto).

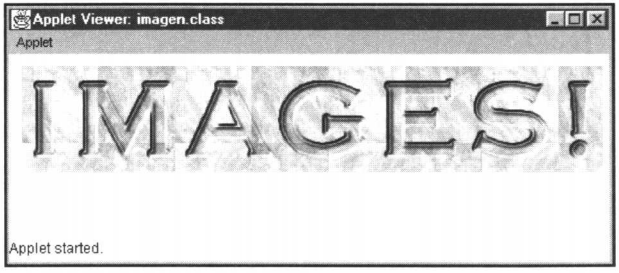


Figura 9.4. Visualizar una imagen.

Redimensionar imágenes

El programador novato está trabajando en un programa de gráficos y necesita su ayuda. "Quiero que el usuario pueda redimensionar imágenes",

dice NP. "¿Cómo se hace?" "No hay problema", le dice. "Sólo hay que especificar la nueva altura y anchura de la imagen en el método *drawImage*".

Para redimensionar una imagen, se puede usar esta versión del método *drawImage* de la clase *Graphics*, sobrecargado, que permite especificar la anchura y la altura de una imagen:

```
drawImage(Image img, int x, int y, int anchura, int altura, ImageObserver
observer)
```

Aquí hay un ejemplo. En esta *applet*, sólo se necesita presionar el ratón en un punto y soltarlo en otro; la *applet* dibujará la imagen que se vio en el punto anterior, redimensionada de forma que ocupe el rectángulo que se ha definido:

```
import java.awt.*;
import java.lang.Math;
import java.awt.event.*;
import java.applet.Applet;

/*
  <APPLET
      CODE=resizer.class
      WIDTH=600
      HEIGHT=300 >
  </APPLET>
*/

public class resizer extends Applet implements MouseListener
{
   Image image;
   boolean mouseUp = false;
   Point start, end;

   public void init()
   {
       image = getImage(getDocumentBase(), "image.jpg");
       addMouseListener(this);
   }

   public void mousePressed(MouseEvent e)
   {
       mouseUp = false;
       start = new Point(e.getX(), e.getY());
   }

   public void mouseReleased(MouseEvent e)
   {
       mouseUp = true;
       end = new Point(Math.max(e.getX(), start.x),
             Math.max(e.getY(), start.y));
       start = new Point(Math.min(e.getX(), start.x),
             Math.min(e.getY(), start.y));
```

```
            repaint();
        }

        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

        public void paint (Graphics g)
        {
            if(mouseUp) {
                int width = end.x - start.x;
                int height = end.y - start.y;
             g.drawImage(image, start.x, start.y, width, height, this);
            }
        }
    }
```

El resultado de este código se muestra en la figura 9.5, donde se puede ver que se ha redimensionado la imagen mostrada en el punto anterior.

Hemos utilizado algunas variaciones del método *drawImage*, que forma parte de la clase *Graphics*. Ahora, es el momento de echar un vistazo a esta clase en sí misma y a todo lo que contiene (gran cantidad de métodos). Veamos el siguiente apartado para los detalles.



Figura 9.5. Redimensionar una imagen.

Dibujar gráficos

El gran jefe aparece envuelto en el humo del cigarro y dice, "El equipo de diseño tiene preparado un programa ganador que quiero que escriba". "¿De qué se trata?", le pregunta. "Permite al usuario dibujar rectas, rectángulos y óvalos, así como dibujo libre con el ratón", dice GJ. "Realmente es una bomba", le responde.

El núcleo de los gráficos AWT está formado por la enorme clase *Graphics*, que se deriva directamente de *java.lang.Object*. El constructor de esta clase se encuentra en la tabla 9.17 y sus métodos en la 9.18.

Tabla 9.17. Constructor de la clase *Graphics*.

Constructor	Descripción
protected Graphics()	Crea un nuevo objeto *Graphics*.

Tabla 9.18. Métodos de la clase *Graphics*.

Método	Descripción
abstract void clearRect(int x, int y, int anchura, *int* altura)	Limpia un rectángulo, rellenándolo con el color de fondo.
abstract void clipRect(int x, int y, int anchura, *int* altura)	Intersecta la región actual cortada con el rectángulo.
abstract void copyArea(int x, int y, int anchura, *int* altura, *int dx, int dy)*	Copia un área del componente a una región dada por dx y dy.
abstract Graphics create()	Crea un nuevo objeto *Graphics*, que es una copia de éste.
Graphics create(int x, int y, int achura, *int* altura)	Crea un nuevo objeto *Graphics* basado en éste, con una nueva área de traslación y de corte.
abstract void dispose()	Se deshace de este contexto gráfico.
void draw3Drect(int x, int y, int anchura, *int* altura, *boolean raised)*	Visualiza un contorno resaltado de 3D de un rectángulo.
abstract void drawArc(int x, int y, int anchura, *int* altura, *int startAngle, int arcAngle)*	Visualiza el contorno de un arco de círculo o elipse.
void drawBytes(byte[] data, int offset, int longitud, *int x, int y)*	Visualiza el texto dado por el *array* de bytes.
void drawChars(char[] data, int offset, int longitud, *int x, int y)*	Visualiza el texto dado por el *array* de caracteres.
abstract boolean drawImage(Image img, int x, int y, Color bgcolor, Image Observer observer)	Muestra la parte de la imagen dada que está disponible.
abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)	Muestra la parte de la imagen dada que está disponible.

Método	Descripción
abstract boolean drawImage(Image img, int x, int y, int anchura, *int* altura, *Color bgcolor, ImageObserver observer)*	Muestra la parte de la imagen que ha sido puesta en escala para que quepa en el interior del rectángulo.
abstract boolean drawImage(Image img, int x, int y, int anchura, *int* altura, *ImageObserver observer)*	Muestra la parte de la imagen que ha sido puesta en escala para que quepa en el interior del rectángulo.
abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)	Muestra la parte de la imagen que está disponible, poniéndola en escala para que quepa en el interior del área dada.
abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)	Muestra la parte de la imagen que está disponible, poniéndola en escala para que quepa en el interior del área dada de la superficie de destino.
abstract void drawLine(int x1, int y1, int x2, int y2)	Muestra una recta, usando el color actual, entre los puntos (x1, y1) y (x2, y2).
abstract void drawOval(int x, int y, int anchura, *int* altura)	Muestra el contorno de un óvalo.
abstract void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)	Muestra un polígono cerrado definido por los *arrays* de coordenadas x e y.
void drawPolygon(Polygon p)	Muestra el contorno de un polígono definido por el objeto *Polygon* dado.
abstract void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)	Muestra una secuencia de líneas unidas definidas por los *arrays* de coordenadas x e y.
void drawRect(int x, int y, int anchura, *int* altura)	Muestra el contorno de un rectángulo dado.
abstract void drawRoundRect(int x, int y, int anchura, *int* altura, *int arcWidth, int arcHeight)*	Muestra el contorno de un rectángulo con las esquinas redondeadas.
abstract void drawString(AttributedCharacterIterator iterator, int x, int y)	Muestra el texto dado por el iterador especificado.
abstract void drawString(String str, int x, int y)	Muestra el texto dado por la cadena especificada.

Método	Descripción
void fill3Drect(int x, int y, int anchura, *int* altura, *boolean raised)*	Pinta un rectángulo 3D resaltado, relleno con el color actual.
abstract void fillArc(int x, int y, int anchura, *int* altura, *int startAngle, int arcAngle)*	Rellena un arco de círculo o elipse cubriendo el rectángulo dado.
abstract void fillOval(int x, int y, int anchura, *int* altura*)*	Rellena el contorno de un óvalo del rectángulo dado, con el color actual.
abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)	Rellena un polígono cerrado definido por los *arrays* de coordenadas x e y.
void fillPolygon(Polygon p)	Rellena el polígono definido por el objeto *Polygon* con el color actual del contexto gráfico.
abstract void fillRect(int x, int y, int anchura, *int* altura)	Rellena el rectángulo dado.
abstract void fillRoundRect(int x, int y, int anchura, *int* altura, *int arcWidth, int arcHeight)*	Rellena el rectángulo con las esquinas redondeadas.
void finalize()	Gestiona la colección *garbage* de este contexto gráfico.
abstract Shape getClip()	Obtiene el área recortada.
abstract Rectangle getClipBounds()	Obtiene el rectángulo que limita el área recortada.
Rectangle getClipBounds(Rectangle r)	Obtiene el rectángulo que limita el área recortada.
Rectangle getClipRect()	Obsoleto. Reemplazado por *getClipBounds()*.
abstract Color getColor()	Obtiene el color actual del contexto gráfico.
abstract Font getFont()	Obtiene la fuente.
FontMetrics getFontMetrics()	Obtiene las métricas de la fuente.
abstract FontMetrics getFontMetrics (Font f)	Obtiene las métricas de la fuente dada.
boolean hitClip(int x, int y, int anchura, *int* altura)	Devuelve verdadero si el área rectangular intersecta al rectángulo del área recortada.

Método	Descripción
abstract void setClip(int x, int y, int achura, *int* altura)	Fija el área recortada actual al rectángulo dado por las coordenadas especificadas.
abstract void setClip(Shape clip)	Devuelve el área recortada de forma arbitraria.
abstract void setColor(Color c)	Establece el color del contexto gráfico al color dado.
abstract void setFont(Font fuente)	Establece la fuente del contexto gráfico a la fuente dada.
abstract void setPaintMode()	Establece el modo de pintar para sobrescribir el destino con el color actual de este contexto gráfico.
abstract void setXORMode(Color c1)	Establece el modo de pintar a alternar entre el color actual de este contexto gráfico y el nuevo color dado, usando XOR.
String toString()	Obtiene un objeto tipo *String* para representar este objeto *Graphics.*
abstract void translate(int x, int y)	Traslada el origen del contexto gráfico al punto (x, y).

Aquí, vamos a utilizar la clase *Graphics* para crear el programa que el gran jefe quería, un programa de gráficos que permita al usuario dibujar rectas, óvalos, rectángulos, rectángulos redondeados y dibujo libre con el ratón, como se muestra en la figura 9.6.

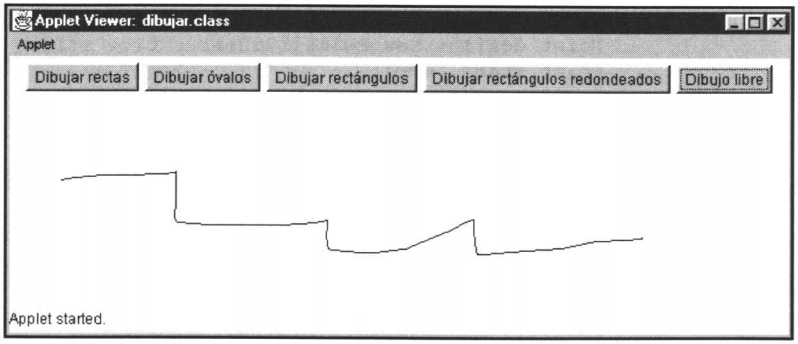


Figura 9.6. Dibujo libre con el ratón.

Este programa se llama dibujar.java en el CD, y así es como funciona: el usuario hace clic sobre un botón indicando qué clase de figura quiere dibujar, con lo que se activa un indicador booleano en el programa. Cuando después el usuario presiona el ratón en la superficie de dibujo, ésa ubicación se almacena como *start,* usando un objeto *Point* (que tiene dos miembros de datos: x e y). Cuando se suelta el ratón en una nueva posición, esta se almacena como *end.* Al soltar el ratón, también se vuelve a pintar el programa, y el usuario puede seleccionar la figura a dibujar, una recta, óvalo, rectángulo o rectángulo redondeado, entre las posiciones *start* y *end* basándose en el indicador boleano fijado al hacer clic sobre los botones.

El dibujo libre con el ratón es diferente. En ese caso, se almacenan hasta 1.000 puntos sobre los que se mueve el ratón. En el momento de dibujar, basta con unir los puntos con líneas (observe que no se genera un evento de ratón por cada pixel sobre el que se mueve el ratón, por lo tanto es necesario dibujar líneas entre las distintas posiciones del ratón que Java comunica).

Así es dibujar.java (echaremos un vistazo a las secciones de dibujos con más detalle en las siguiente páginas):

```
import java.awt.*;
import java.lang.Math;
import java.awt.event.*;
import java.awt.Graphics;
import java.applet.Applet;

/*
  <APPLET
      CODE=dibujar.class
      WIDTH=600
      HEIGHT=200 >
  </APPLET>
*/

  public class draw extends Applet implements ActionListener, MouseListener,
MouseMotionListener {
      Button bDraw, bLine, bOval, bRect, bRounded;
      Point dot[] = new Point[1000];
      Point start, end;
      int dots = 0;

      boolean mouseUp = false;
      boolean draw = false;
      boolean line = false;
      boolean oval = false;
      boolean rectangle = false;
      boolean rounded = false;

      public void init()
      {
          bLine = new Button("Dibujar rectas");
```

```
        bOval = new Button("Dibujar óvalos");
        bRect = new Button("Dibujar rectángulos");
        bRounded = new Button("Dibujar rectángulos redondeados");
        bDraw = new Button("Dibujo libre");

        add(bLine);
        add(bOval);
        add(bRect);
        add(bRounded);
        add(bDraw);

        bLine.addActionListener(this);
        bOval.addActionListener(this);
        bRect.addActionListener(this);
        bRounded.addActionListener(this);
        bDraw.addActionListener(this);

        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mousePressed(MouseEvent e)
    {
        mouseUp = false;
        start = new Point(e.getX(), e.getY());
    }

    public void mouseReleased(MouseEvent e)
    {
        if(line){
            end = new Point(e.getX(), e.getY());
        } else {
            end = new Point(Math.max(e.getX(), start.x),
                Math.max(e.getY(), start.y));
            start = new Point(Math.min(e.getX(), start.x),
                Math.min(e.getY(), start.y));
        }
        mouseUp = true;
        repaint();
    }

    public void mouseDragged(MouseEvent e)
    {
        if(draw){
            dot[dots] = new Point(e.getX(), e.getY());
            dots++;
            repaint();
        }
    }

    public void mouseClicked(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
```

```
    public void paint (Graphics g)
    {
        if (mouseUp) {
            int width = end.x - start.x;
            int height = end.y - start.y;

            if(line){
                g.drawLine(start.x, start.y, end.x, end.y);
            }
            else if(oval){
                g.drawOval(start.x, start.y, width, height);
            }
            else if(rectangle){
                g.drawRect(start.x, start.y, width, height);
            }
            else if(rounded){
                 g.drawRoundRect(start.x, start.y, width, height, 10,
10);
            }
            else if(draw){
                     for(int loop_index = 0; loop_index < dots - 1;
loop_index++){
                    g.drawLine(dot[loop_index].x, dot[loop_index].y,
                        dot[loop_index + 1].x, dot[loop_index + 1].y);
                }
            }
        }
    }

    public void actionPerformed(ActionEvent e)
    {
        setFlagsFalse();
        if(e.getSource() == bDraw)draw = true;
        if(e.getSource() == bLine)line = true;
        if(e.getSource() == bOval)oval = true;
        if(e.getSource() == bRect)rectangle = true;
        if(e.getSource() == bRounded)rounded = true;
    }

    void setFlagsFalse()
    {
        rounded = false;
        line = false;
        oval = false;
        rectangle = false;
        draw = false;
    }
}
```

Así es dibujar.java. Ahora, veremos algunas de sus funciones de dibujo. Todas estas funciones, excepto la de dibujo libre, permiten dibujar una figura entre las posiciones *start* y *end*, que el usuario indica al arrastrar el ratón.

Dibujar rectas

Usando el objeto *Graphics,* se puede dibujar una recta entre los puntos (x1, y1) y (x2, y2) con el método *drawLine*:

```
drawLine(int x1, int y1, int x2, int y2);
```

Así es como aparecía en dibujar.java:

```
g.drawLine(start.x, start.y, end.x, end.y);
```

Se puede ver el resultado de dibujar.java en la figura 9.7.

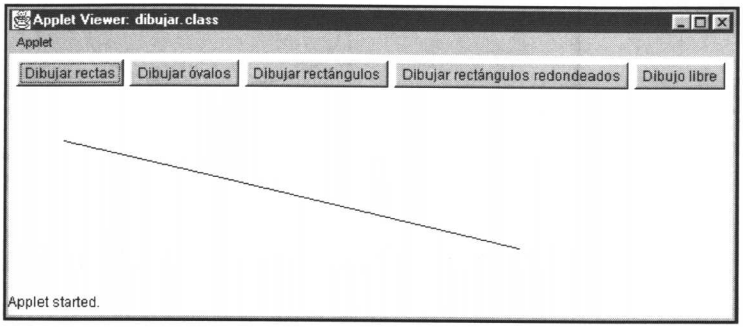


Figura 9.7. Dibujar una recta con el ratón.

Dibujar óvalos

Las elipses, incluyendo los círculos, se denominan óvalos en AWT, y se pueden dibujar con el método *drawOval* de la clase *Graphics:*

```
drawOval(int x, int y, int width, int height);
```

Así es cómo se dibujan los óvalos cuando se ejecuta dibujar.java:

```
int width = end.x - start.x;
int height = end.y - start.y;
g.drawOval(start.x, start.y, width, height);
```

Se puede ver el resultado de dibujar.java, en la figura 9.8.

Dibujar rectángulos

Se pueden dibujar rectángulos utilizando el método *drawRect* de la clase *Graphics:*

```
drawRect(int x, int y, int width, int height);
```

Así se hace en dibujar.java:

```
int width = end.x - start.x;
int height = end.y - start.y;
g.drawRect(start.x, start.y, width, height);
```

El resultado de dibujar.java se muestra en la figura 9.9.

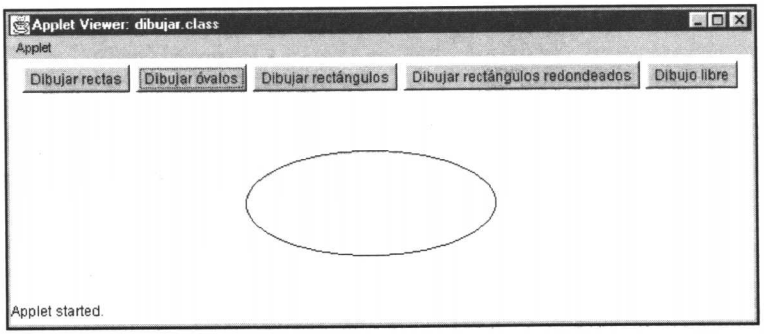


Figura 9.8. Dibujar un óvalo con el ratón.

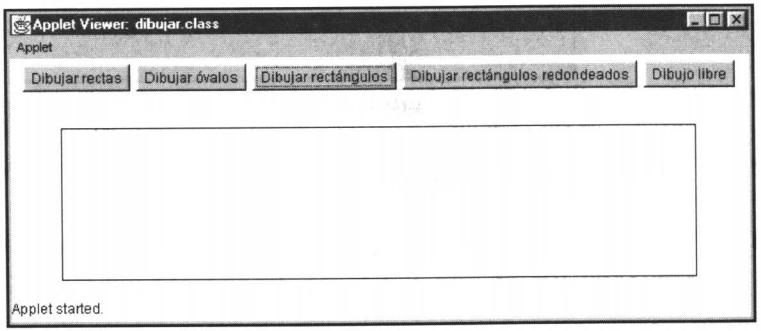


Figura 9.9. Dibujar un rectángulo con el ratón.

Dibujar rectángulos redondeados

Los rectángulos redondeados (es decir, rectángulos con las esquinas redondeadas) se pueden dibujar utilizando el método *drawRoundRect* de la clase *Graphics:*

```
drawRoundRect(int x, int y, int width, int height, int arcWidth, int
arcHeight);
```

A continuación, se especifican la anchura y la altura del arco, en pixels, lo cual especifica cuánto hay que redondear las esquinas. Así es como se crea un rectángulo redondeado en dibujar.java:

```
int width = end.x - start.x;
int height = end.y - start.y;
g.drawRoundRect(start.x, start.y, width, height, 10, 10);
```

Se puede ver el resultado de dibujar.java en la figura 9.10.

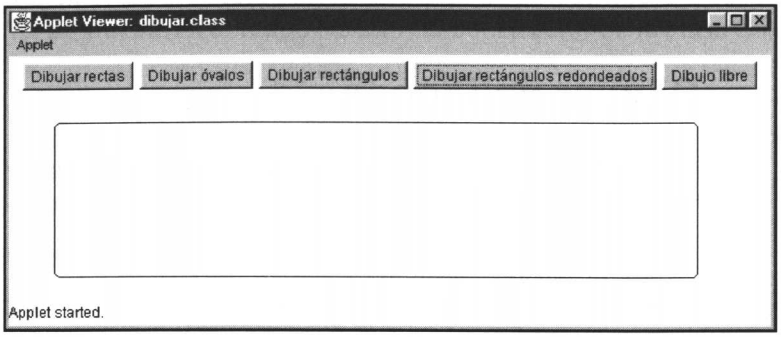


Figura 9.10. Dibujar un rectángulo redondeado con el ratón.

Dibujo libre

El dibujo libre con el ratón se puede hacer utilizando la clase AWT *Graphics,* pero hay que implementarlo en el código. Así es como se hizo en dibujar.java usando el método *mouseDragged:* una vez que se está seguro de que el usuario está dibujando libremente, comprobando que el indicador *draw* es verdadero, se salvan todas las posiciones del ratón en un *array* llamado *dot[ ]* al arrastrarlo:

```
public void mouseDragged(MouseEvent e)
    {
        if(draw){
            dot[dots] = new Point(e.getX(), e.getY());
            dots++;
            repaint();
        }
    }
```

Después, en el momento de dibujar la figura, se unen los puntos usando rectas, como sigue:

```
for(int loop_index = 0; loop_index < dots - 1; loop_index++){
    g.drawLine(dot[loop_index].x, dot[loop_index].y,
    dot[loop_index + 1].x, dot[loop_index + 1].y);
}
```

El resultado se muestra en la figura 9.6 (ya visto anteriormente).

Dibujar arcos

Con el método *drawArc* de la clase *Graphics* se pueden dibujar arcos (se especifica el ángulo en grados):

```
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle);
```

Dibujar polígonos

Hay gran cantidad de formas de dibujar polígonos y rectas de múltiples segmentos con AWT:

```
drawPolygon(int[] xPoints, int[] yPoints, int nPoints);
drawPolygon(Polygon p);
drawPolyline(int[] xPoints, int[] yPoints, int nPoints);
```

Establecer los modos de dibujo

AWT también permite alternar entre dos modos de dibujo, modo directo y modo XOR, con estos métodos:

```
setXORMode(Color c1);
setPaintMode();
```

En el modo directo, cualquier cosa que se pinte sólo cubre lo que esté debajo, pero en modo XOR, lo que se dibuje será exclusivamente *ORed* con un color particular que ya está en la pantalla. Es muy útil, porque cuando se hace XOR A con B dos veces, B se almacena, lo que significa que se puede dibujar algo en la pantalla y después dibujarlo usando el modo XOR. A continuación, se recupera cualquier cosa que estuviera originalmente en la pantalla. Por ejemplo, si se quiere permitir al usuario que estire, de forma interactiva, las figuras que dibuja en dibujar.java, se haría como sigue: el usuario dibuja una figura; después, cuando mueve el ratón, se redibuja la figura usando el modo XOR para borrarla. A continuación se vuelve a dibujar con su nuevo tamaño.

Seleccionar colores

"Mi trabajo con los gráficos ha sido un poco monótono", dice el programador novato, "porque todo es de color negro". "Bien", le contesta, "puede seleccionar el color fácilmente".

Ninguna lección sobre gráficos estaría completa sin hablar de los colores, que se gestionan con la clase *Color* de AWT. Los campos de esta clase se encuentran en la tabla 9.19, sus constructores en la tabla 9.20 y sus métodos en la 9.21.

Por ejemplo, para crear un nuevo color, se pueden especificar los valores rojo, verde y azul (en el rango 0 a 255) al constructor de la clase *Color*, como sigue:

```
Color c = new Color(red, green, blue);
```

Además, hay libertad para fijar el nuevo color de dibujo, usando *setForeground:*

```
setForeground(c);
```

Ahora, las operaciones de dibujo tendrán lugar en el color que se especifique. Además se puede usar un color predefinido, como *Color.blue,* en este caso, donde se está fijando el color de fondo:

```
setBackground(Color.blue);
```

De hecho, también se pueden rellenar figuras usando el color que se especifique con los métodos de *Graphics,* como *fillArc, fillOval,* etc.

Tabla 9.19. Campos de la clase *Color.*

Campo	Descripción
static Color black	Color negro.
static Color blue	Color azul.
static Color cyan	Color cyan.
static Color darkGray	Color gris oscuro.
static Color gray	Color gris.
static Color green	Color verde.
static Color lightGray	Color gris claro.
static Color magenta	Color magenta.
static Color orange	Color naranja.
static Color pink	Color rosa.
static Color red	Color rojo.
static Color white	Color blanco.
static Color yellow	Color amarillo.

Tabla 9.20. Constructores de la clase *Color.*

Constructor	Descripción
Color(ColorSpace cspace, float[] components, float alpha)	Crea un color en el espacio suministrado por *ColorSpace.*
Color(float r, float g, float b)	Crea un color opaco con los valores dados para rojo, verde y azul, en el rango 0.0 a 1.0.
Color(float r, float g, float b, float a)	Crea un color opaco con los valores dados para rojo, verde, azul y alfa, en el rango 0.0 a 1.0.
Color(int rgb)	Crea un color opaco con el valor RGB combinado, que consiste en el valor rojo en los bits 16-23, el verde en 8-15 y el azul en los bits 0-7.
Color (int rgba, boolean hasalpha)	Crea un color con el valor RGBA combinado, que consiste en el valor de alfa en los bits 24-31, rojo en los bits 16-23, el verde en 8-15 y el azul en los bits 0-7.
Color(int r, int g, int b)	Crea un color opaco con los valores rojo, azul y verde dados, en el rango 0 a 255.
Color(int r, int g, int b, int a)	Crea un color opaco con los valores rojo, azul, verde y alfa dados, en el rango 0 a 255.

Tabla 9.21. Métodos de la clase *Color.*

Método	Descripción
Color brighter()	Hace una versión del color con más brillo.
PaintContext createContext(ColorModel cm, rectangle r, Rectangle2D r2d, AffineTransform xform, Rendering Hints hints)	Crea y devuelve un contexto para generar un modelo en color sólido.
Color darker()	Hace una versión del color más oscura.
static Color decode(String nm)	Convierte una cadena en un entero y devuelve ese color.

Método	Descripción
boolean equals(Object obj)	Determina si otro color es igual a éste.
int getAlpha()	Obtiene el valor alfa.
int getBlue()	Obtiene el valor azul.
static Color getColor(String nm)	Busca un color en las propiedades del sistema.
static Color getColor(String nm, Color v)	Busca un color en las propiedades del sistema.
static Color getColor(String nm, int v)	Busca un color en las propiedades del sistema.
float[] getColorComponents(ColorSpace cspace, float[] compArray)	Obtiene un *array* de *float* que contiene los componentes de color del objeto *Color* en el *color space.*
float[] getColorComponents(float[] compArray)	Obtiene un *array* de *float* que contiene los componentes de color (no alfa) del objeto *Color.*
ColorSpace getColorSpace()	Obtiene el *color space* del objeto *Color.*
float[] getComponents(ColorSpace cspace, float[] compArray)	Obtiene un *array* de *float* que contiene los componentes de color y alfa del objeto *Color* en el *color space.*
float[] getComponents(float[] compArray)	Obtiene un *array* de *float* que contiene los componentes de color y alfa del objeto *Color.*
int getGreen()	Obtiene el valor verde.
static Color getHSBColor(float h, float s, float b)	Crea un objeto *Color* basado en los valores HSB.
int getRed()	Obtiene el valor rojo.
int getRGB()	Obtiene el valor RGB que representa el color en el modelo de color por defecto.
float[] getRGBColorComponents (float[] comArray)	Obtiene un *array* de *float* que contiene los componentes de color (no alfa) del objeto *Color* en el *color space.*

Método	Descripción
float[] getRGBColorComponents (float[] comArray)	Obtiene un *array* de *float* que contiene los componentes de color y alfa del objeto *Color* en el *color space.*
int getTransparency()	Devuelve el modo de transparencia para este color.
int hashCode()	Calcula el *hashcode* paraeste color.
static int HSBTOrgb(float hue, float brightness)	Convierte los componentes de un color dado por el modelo HSB al modelo RGB por defecto.
static float[] RGBtoHSB(int r, int g, int b, float[] hsvals)	Convierte los componentes de un color, dado según el modelo RGB, a un conjunto de valores equivalente según el modelo HSB.
String toString()	Obtiene una representación en cadena de este color.

Usar *Canvases*

El componente *canvas* se construye especialmente para soportar las operaciones de gráficos.

Como su nombre indica, proporciona un espacio en blanco para dibujar, usando el objeto *Graphics* que se le pasa al método *paint*. Este es el diagrama de herencia de la clase AWT *Canvas:*

```
java.lang.Object
|____java.util.EventObject
     |____java.awt.Component
          |____java.awt.Canvas
```

Los constructores de esta clase se encuentran en la tabla 9.22 y sus métodos en la 9.23.

Tabla 9.22. Constructores de la clase *Canvas.*

Constructor	Descripción
Canvas()	Crea un nuevo *canvas.*
Canvas(GraphicsConfiguration config)	Crea un nuevo *canvas,* dado un objeto *GraphicsConfiguration.*

Tabla 9.23. Métodos de la clase *Canvas.*

Método	Descripción
void addNotify()	Crea el compañero del *canvas.*
void paint(Graphics g)	Se le llama para volver a pintar el *canvas.*

Los *canvases* se usan normalmente para soportar una forma básica de animación, ya que se puede utilizar el método *setLocation* de la clase *Component* para mover un *canvas* (o cualquier otro componente). Aquí hay un a*pplet* que hace esto cuando se hace clic sobre ella:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/*
  <APPLET
      CODE=canvas.class
      WIDTH=400
      HEIGHT=200 >
  </APPLET>
*/

public class canvas extends java.applet.Applet implements MouseListener
{

    graphicsCanvas gc;
    Button button1;

    public void init()
    {
        gc = new graphicsCanvas();
        gc.setSize(100, 100);
        add(gc);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent e){}

    public void mouseClicked(MouseEvent e)
    {
       for(int loop_index = 0; loop_index < 150; loop_index++){
           gc.setLocation(loop_index, 0);
       }
    }
    public void mouseReleased(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
}

class graphicsCanvas extends java.awt.Canvas
```

```
{
    public void paint (Graphics g)
    {
        g.drawOval(10, 50, 40, 40);
        g.drawLine(10, 50, 50, 90);
        g.drawLine(50, 50, 10, 90);
    }
}
```

El resultado se muestra en la figura 9.11. Cuando se hace clic sobre la *applet,* el *canvas,* que visualiza una figura pequeña, se mueve a la izquierda y luego hacia la derecha.

Este ejemplo se encuentra en el CD como canvas.java. Observe que, como al método *paint* de un *canvas* se le pasa un objeto *Graphics,* se puede usar cualquier método *Graphics* en un *canvas.*

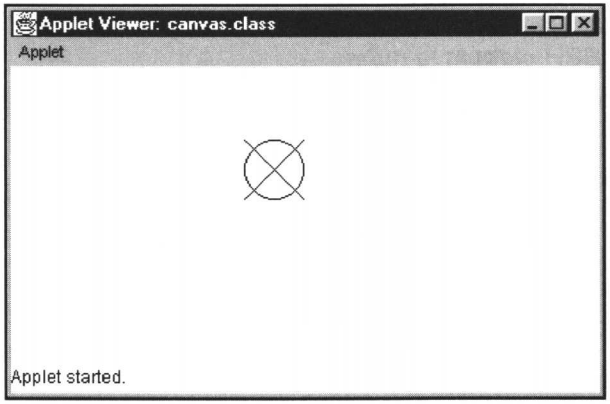


Figura 9.11. Animación sencilla con un *canvas.*

Usar la interfaz *ImageObserver*

El especialista en soporte de productos (ESP) aparece y dice tristemente, "Ha habido quejas de cómo su programa dibuja las imágenes que se descargan de la red Internet, primero sólo aparece parte de la imagen, y el resto, gradualmente". "Así funciona la red Internet", le responde, sorprendido. "¿No se puede arreglar?", pregunta ESP.

Para ver las imágenes que se cargan se pueden usar los métodos de la interfaz *ImageObserver.* Como vimos antes, es necesario indicar a la interfaz *ImageObserver* que se carga una imagen en un *applet* y especificar la implementación por defecto de esta interfaz. En este punto, crearemos nues-

tra propia implementación de esta interfaz. La interfaz *ImageObserver* sólo tiene un método, *imageUpdate:*

```
boolean imageUpdate(image img, int infoflags, int x, int y, int width,
int height)
```

A este método se le llama cuando la información de una imagen, que está siendo cargada de forma asíncrona, está disponible. Estos son los *flags* que se pasan en *infoflags* en el método *imageUpdate:*

- *ABORT:* Indica que una imagen que se estaba siguiendo fue abortada.
- *ALLBITS:* Indica que una imagen estática que fue dibujada previamente ya está completa.
- *ERROR:* Indica que una imagen que se estaba siguiendo ha producido un error.
- *FRAMEBITS:* Indica que otro marco completo de una imagen está disponible para dibujarse.
- *HEIGHT:* Indica que la altura de la imagen base está disponible (y se puede leer del argumento *height* del método *imageUpdate).*
- *PROPERTIES:* Indica que las propiedades de la imagen están disponibles.
- *SOMEBITS:* Indica que hay disponibles más pixels para dibujar la imagen.
- *WIDTH:* Indica que la anchura de la imagen base está disponible (y se puede leer del argumento *width* del método *imageUpdate).*

El método *imageUpdate* devuelve *verdadero* si se necesitan actualizaciones; devuelve *false* si se ha recibido la información que se quiere.

Este es un ejemplo. En este caso, se sobrescribe el método *imageUpdate* para llamar a *repaint* y visualizar una imagen, pero sólo cuando está totalmente cargada:

```
import java.awt.*;
import java.applet.*;

/*
  <APPLET
      CODE=iobserver.class
      WIDTH=600
      HEIGHT=150 >
  </APPLET>
*/

public class iobserver extends Applet
{
```

```
    Image image;

    public void init()
    {
        image = getImage(getDocumentBase(), "image.jpg");
    }

    public void paint(Graphics g)
    {
        g.drawImage(image, 10, 10, this);
    }

    public boolean imageUpdate(Image img, int flags, int x, int y, int w,
int h)
    {
        if ((flags & ALLBITS) != 0) {
            repaint(x, y, w, h);
        }
        return (flags & ALLBITS) == 0;
    }
}
```

El resultado se muestra en la figura 9.12, donde la imagen aparece una vez que ha sido totalmente cargada. Este ejemplo está en el CD como *iobserver.java.* Los programadores han comentado a *Sun* que la interfaz *ImageObserver* es demasiado compleja, especialmente cuando se trata de descargas múltiples. Por ello, *Sun* creó la clase *MediaTracker* (ver el siguiente apartado).

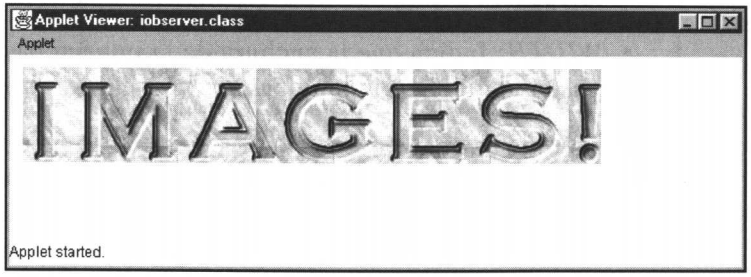


Figura 9.12. Usar la interfaz *ImageObserver.*

Usar la clase *MediaTracker*

La clase *MediaTracker* proporciona una manera fácil (más fácil que utilizar los objetos *ImageObserver)* para monitorizar la descarga de imágenes. Para empezar, sólo pasaremos al método *addImage,* una imagen y un ID que se quiera utilizar para esa imagen (se puede usar el mismo ID que se utilice para enumerar las imágenes si se quieren controlar como si fueran un grupo):

```
void addImage(Image image, int id)
void addImage(Image image, int id, int w, int h)
```

Con el método *checkId* se puede verificar si una imagen o grupo de imágenes han terminado de cargarse, como sigue:

```
boolean checkID(int id)
boolean checkID(int id, boolean load)
```

Este método devuelve *Verdadero* si la imagen o grupo de imágenes involucradas han terminado de cargarse; en caso contrario, devuelve *False*. Además se puede usar el método *waitForAll;* este método vuelve cuando todas las imágenes rastreadas están cargadas.

Los campos de la clase *MediaTracker* se encuentran en la tabla 9.24, su constructor en la 9.25 y sus métodos en la tabla 9.26.

Tabla 9.24. Campos de la clase *MediaTracker.*

Campos	Descripción
static int ABORTED	Indica que la descarga se abortó.
static int COMPLETE	Indica que se completó la descarga del medio.
static int ERRORED	Indica que hubo un error en la descarga de algún medio desarrollado.
static int LOADING	Indica que algún dato se está cargando actualmente.

Tabla 9.25. Constructor de la clase *MediaTracker.*

Constructor	Descripción
MediaTracker(Component comp)	Crea un *media tracker.*

Tabla 9.26. Métodos de la clase *MediaTracker.*

Método	Descripción
void addImage(Image image, int id)	Añade una imagen a las imágenes que están siendo controladas.
void addImage(Image image, int id, int w, int h)	Añade una imagen a escala, a la lista de imágenes que están siendo controladas.
boolean checkAll(boolean load)	Verificación para ver si todas las imágenes que se están siguiendo se han terminado de cargar.

Método	Descripción
boolean checkID(int id)	Verificación para ver si todas las imágenes que se están siguiendo y que tienen el identificador dado, se han terminado de cargar.
boolean checkID(int id, boolean load)	Verificación para ver si todas las imágenes que se están siguiendo y que están etiquetadas con el identificador dado, se han terminado de cargar.
Object[] getErrorsAny()	Obtiene una lista de todas las imágenes que han dado un error.
Object[] getErrorsID(int id)	Obtiene una lista de las imágenes con el ID dado que han dado un error.
boolean isErrorAny()	Verifica el estado de error de todas las imágenes.
boolean isErrorID(int id)	Verifica el estado de error de todas las imágenes con el identificador dado.
void removeImage(Image image)	Elimina una imagen de este *media tracker.*
void removeImage(Image image, int id)	Elimina la imagen dada con el ID especificado.
void removeImage(Image image, int id, int anchura, int altura)	Elimina la imagen que tiene la anchura, altura e ID dados.
int statusAll(boolean load)	Obtiene el desplazamiento de bits OR del estado de todos los *media tracker.*
int statusID(int id, boolean load)	Obtiene el desplazamiento de bits OR del estado de todos los *media tracker* con el identificador dado.
void waitForAll()	Inicia la carga de todas las imágenes.
boolean waitForAll(long ms)	Inicia la carga de todas las imágenes.
void waitForID(int id)	Inicia la carga de todas las imágenes del identificador dado.
boolean waitForID(int id, long ms)	Inicia la carga de todas las imágenes del identificador dado.

Veamos un ejemplo. En este caso, se usa el método *waitForAll* para esperar a que una imagen esté totalmente cargada (observe las sentencias *try/*

catch, que son las que gestionan las excepciones; las veremos con más detalle más adelante):

```
import java.awt.*;
import java.applet.*;

/*
  <APPLET
      CODE=mediatracker.class
      WIDTH=600
      HEIGHT=150 >
  </APPLET>
*/

public class mediatracker extends Applet
{
    Image image;

    public void init()
    {
        MediaTracker tracker = new MediaTracker(this);
        image = getImage(getDocumentBase(), "image.jpg");
        tracker.addImage(image, 0);
        try {
            tracker.waitForAll();
        }
        catch (InterruptedException ex) { }
  }

    public void paint(Graphics g)
    {
        g.drawImage(image, 10, 10, this);
    }
}
```

El resultado se muestra en la figura 9.13. Este ejemplo está en el CD como *mediatracker.java*.

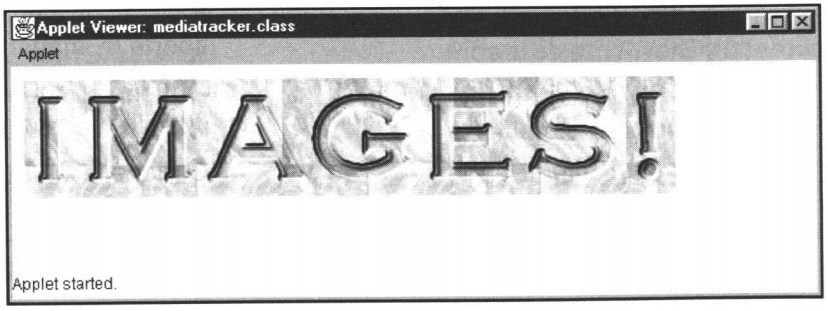


Figura 9.13. Usar el objeto *MediaTracker*.

Trabajar pixel por pixel: Las clases *PixelGrabber* y *MemoryImageSource*

"De acuerdo", dice el programador novato, "hay algunas cosas que quiero hacer gráficamente y sólo lo puedo hacer si tengo acceso directo a los pixels de la imagen. No sé utilizar Java para eso". "Por supuesto que sabe", le contesta. PN dice: "¡Explíqueme más cosas!"

Para situar los pixels de una imagen en un *array* y acceder a los mismos, se pueden usar los objetos *PixelGrabber*. Este es el diagrama de herencia de *PixelGrabber:*

```
java.lang.Object
|____java.awt.image.PixelGrabber
```

Los constructores de esta clase se encuentran en la tabla 9.27 y sus métodos en la 9.28.

Tabla 9.27. Constructores de la clase *PixelGrabbers.*

Constructor	Descripción
PixelGrabber(Image img, int x, int y, int w, int h, boolean forceRGB)	Crea un objeto *PixelGrabber* para coger la sección (x, y, w, h) de pixels.
PixelGrabber(Image img, int x, int y, int w, int h, int[] pix)	Crea un objeto *PixelGrabber* para coger la sección (x, y, w, h) de pixels y situarlos en un *array* de la imagen dada.

Tabla 9.28. Métodos de la clase *PixelGrabber.*

Método	Descripción
void abortGrabbing()	Pide al objeto *PixelGrabber* que aborte la imagen.
ColorModel getColorModel()	Obtiene el modelo de color para las pixels en el *array.*
int getHeight()	Obtiene la altura del buffer de pixel.
Object getPixels()	Obtiene el buffer de pixel.
int getStatus()	Obtiene el estado de los pixels.
int getWidth()	Obtiene la anchura del buffer de pixels.

Método	Descripción
boolean grabPixels()	Solicita la imagen o el productor de imagen para empezar a entregar los pixels.
boolean grabPixels(long ms)	Pide la imagen o el productor de imagen para empezar a entregar los pixels y espera a todos ellos, hasta que finalice el tiempo dado.
void imageComplete(int status)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void setColorModel(ColorModel model)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void setDimensions(int width, int height)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void setHints(int hints)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void setPixels(int srcX, int srcY, intsrcW, int srcH, ColorModel model, byte[] pixels, int srcOff, int srcScan)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void setPixels(int srcX, int srcY, intsrcW, int srcH, ColorModel model, int[] pixels, int srcOff, int srcScan)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void setProperties(Hashtable props)	Parte del API *ImageConsumer*, que esta clase debe implementar para recuperar los pixels.
void startGrabbing()	Pide al objeto *PixelGrabber* que empiece a sacar los pixels.
int status()	Obsoleto. Reemplazado por *getStatus()*.

Para crear una imagen de un *array* de pixels, se puede usar la clase *MemoryImageSource*. El diagrama de herencia de esta clase es:

```
java.lang.Object
|____java.awt.image.MemoryImageSource
```

Los constructores de la clase *MemoryImageSource* se encuentran en la tabla 9.29 y sus métodos en la tabla 9.30.

Tabla 9.29. Constructores de la clase *MemoryImageSource*.

Constructor	Descripción
MemoryImageSource(int w, int h, ColorModel cm, byte[] pix, int off, int scan)	Crea un objeto *ImageProducer* que usa un *array* de bytes.
MemoryImageSource(int w, int h, ColorModel cm, byte[] pix, int off, int scan, Hashtable props)	Crea un objeto *ImageProducer* que usa un *array* de bytes y propiedades.
MemoryImageSource(int w, int h, ColorModel cm, int[] pix, int off, int scan)	Crea un objeto *ImageProducer* que usa un *array* de enteros.
MemoryImageSource(int w, int h, ColorModel cm, byte[] pix, int off, int scan, Hashtable props)	Crea un objeto *ImageProducer* que usa un *array* de enteros y propiedades.
MemoryImageSource(int w, int h, int[] pix, int off, int scan)	Crea un objeto *ImageProducer* que usa un *array* de enteros en el modelo de color RGB por defecto.
MemoryImageSource(int w, int h, int[] pix, int off, int scan, Hashtable props)	Crea un objeto *ImageProducer* que usa un *array* de enteros en el modelo de color RGB por defecto y las propiedades.

Tabla 9.30. Métodos de la clase *MemoryImageSource*.

Método	Descripción
void addConsumer(ImageConsumer ic)	Añade un objeto *ImageConsumer* a la lista de consumidores.
boolean isConsumer(ImageConsumer ic)	Indica si un objeto *ImageConsumer* está en la lista de imágenes de consumidores.
void newPixels()	Envía todo un nuevo buffer de pixels a cualquier objeto *ImageConsumer.*
void newPixels(byte[] newpix, ColorModel newmodel, int offset, int scansize)	Cambia a un nuevo *array* de bytes.

Método	Descripción
void newPixels(int[] newpix, ColorModel newmodel, int offset, int scansize)	Cambia a un nuevo *array* de enteros.
void newPixels(int x, int y, int w, int h)	Envía una región rectangular del *buffer* de pixels a los objetos *ImageConsumer.*
void newPixels(int x, int y, int w, int h, boolean framenotify)	Envía una región rectangular del *buffer* de pixels a los objetos *ImageConsumer.*
void removeConsumer(ImageConsumer ic)	Elimina un objeto *ImageConsumer* de la lista de consumidores.
void requestTopDownLeftRightResend (ImageConsumer ic)	Solicita que los datos de la imagen sean entregados de una vez , en orden de arriba a abajo y de izquierda a derecha.
void setAnimated(boolean animated)	Cambia la imagen en memoria a una animación o a un estado de imagen sencilla.
void setFullBuferUpdates(boolean fullbuffers)	Indica si esta imagen animada debería ser siempre actualizada enviando el buffer completo de pixels.
void startProduction(ImageConsumer ic)	Añade un objeto *ImageConsumer* a la lista de consumidores y empieza la entrega de datos de la imagen.

Este es un ejemplo en el que se usan las clases *PixelGrabber* y *MemoryImageSource.* En este caso, se lee una imagen y se copia en un nuevo objeto imagen. Esto se hace cargando la imagen que se ha utilizado en los ejemplos anteriores de este capítulo, que es de 485 por 88 pixels. Primero, se carga la imagen en *image,* se sitúan sus pixels en un *array* llamado *pixels* usando el método *grabPixels* del objeto *PixelGrabber* y luego se crea una nueva imagen, *image2,* usando el objeto *MemoryImageSource* y el método *createImage* de la clase *Applet:*

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;

/*
  <APPLET
      CODE=copiar.class
```

```
        WIDTH=600
        HEIGHT=150 >
    </APPLET>
*/

public class copiar extends Applet {
    Image image, image2;

    public void init()
    {
        image = getImage(getDocumentBase(), "image.jpg");
        int pixels[] = new int[485 * 88];

      PixelGrabber pg = new PixelGrabber(image, 0, 0, 485, 88, pixels,
0, 485);
        try {
            pg.grabPixels();
        }
        catch (InterruptedException e) {}

        for (int loop_index = 0; loop_index < 485 * 88; loop_index++){
            int p = pixels[loop_index];
            int red = 0xff & (p >> 16);
            int green = 0xff & (p >> 8);
            int blue = 0xff & p;
           pixels[loop_index] = (0xff000000 | red << 16 | green << 8 |
blue);
        }

      image2 = createImage(new MemoryImageSource(485, 88, pixels, 0 ,
485));
    }

    public void paint(Graphics g)
    {
        g.drawImage(image2, 10, 10, this);
    }
}
```

Esto es todo lo que hay que ver aquí, ahora la imagen se ha copiado en *image2* y se ha visualizado. Este ejemplo está en el CD como copiar.java. Ahora que tenemos acceso a los pixels de la imagen, haremos alguna cosa más con las imágenes, como el brillo, que veremos en el siguiente punto.

Dar brillo a las imágenes

Se puede dar brillo a las imágenes incrementando sus valores para el color rojo, verde y azul en la misma cantidad. En el siguiente código, sumaremos 20 a cada valor de color:

```
import java.awt.*;
import java.applet.*;
```

```
import java.awt.image.*;

/*
  <APPLET
      CODE=brillo.class
      WIDTH=600
      HEIGHT=150 >
  </APPLET>
*/

public class brillo extends Applet {
    Image image, image2;

    public void init()
    {
        image = getImage(getDocumentBase(), "image.jpg");
        int pixels[] = new int[485 * 88];

       PixelGrabber pg = new PixelGrabber(image, 0, 0, 485, 88, pixels,
0, 485);
        try {
            pg.grabPixels();
        }
        catch (InterruptedException e) {}

for (int loop_index = 0; loop_index < 485 * 88; loop_index++){
     int p = pixels[loop_index];
     int red = (0xff & (p >> 16)) + 20;
     int green = (0xff & (p >> 8)) + 20;
     int blue = (0xff & p) + 20;
     if (red > 255) red = 255;
     if (green > 255) green = 255;
     if (blue > 255) blue = 255;
          pixels[loop_index] = (0xff000000 | red << 16 | green << 8 |
blue);
}

       image2 = createImage(new MemoryImageSource(485, 88, pixels, 0 ,
485));
    }

    public void paint(Graphics g)
    {
        g.drawImage(image2, 10, 10, this);
    }
}
```

El resultado se muestra en la figura 9.14. Ahora, hemos manipulado las imágenes en el código.

Convertir imágenes a escala de grises

Se pueden convertir imágenes a grises haciendo la media de los valores rojo, verde y azul para cada pixel. En este caso, se convierte una imagen en

color, *whirl.gif* (imagen blanca y roja que se verá más adelante, cuando tratemos la animación de gráficos) a grises. Así es el código:

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;

/*
  <APPLET
      CODE=escaladegrises.class
      WIDTH=300
      HEIGHT=300 >
  </APPLET>
*/

public class escaladegrises extends Applet {
    Image image, image2;

    public void init()
    {
        image = getImage(getDocumentBase(), "whirl1.gif");
        int pixels[] = new int[248 * 248];

      PixelGrabber pg = new PixelGrabber(image, 0, 0, 248, 248, pixels,
0, 248);
        try {
            pg.grabPixels();
        }
        catch (InterruptedException e) {}

for (int loop_index = 0; loop_index < 248 * 248; loop_index++){
   int p = pixels[loop_index];
   int red = (0xff & (p >> 16));
   int green = (0xff & (p >> 8));
   int blue = (0xff & p);
   int avg = (int) ((red + green + blue) / 3);
   pixels[loop_index] = (0xff000000 | avg << 16 | avg << 8 | avg);
}

        image2 = createImage(new MemoryImageSource(248, 248, pixels, 0
, 248));
    }

    public void paint(Graphics g)
    {
        g.drawImage(image2, 10, 10, this);
    }
}
```

El resultado se muestra en la figura 9.15. Por supuesto, ver una imagen en escala de grises en esta figura quizás no sea lo suficientemente convincente en un libro lleno de imágenes en blanco y negro. En su lugar, intente ejecutar el ejemplo del CD, escaladegrises.java.

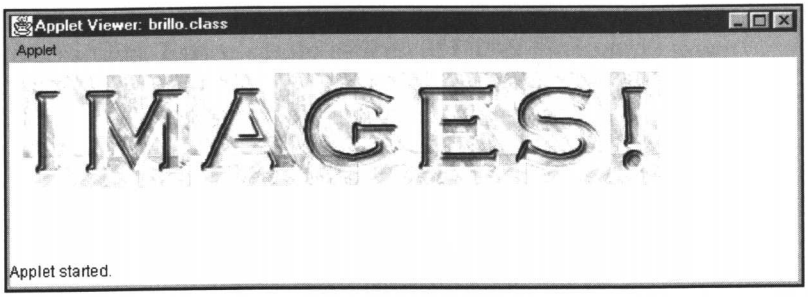


Figura 9.14. Usar el objeto *MediaTracker.*

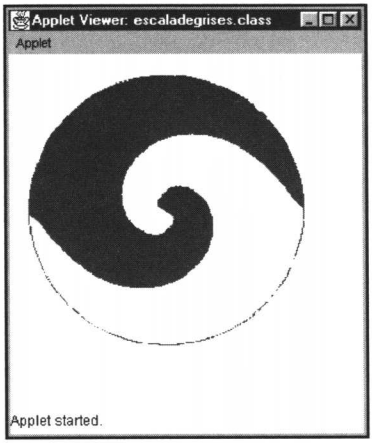


Figura 9.15. Convertir una imagen a grises.

Realzar imágenes

"Realzar" imágenes es un efecto potente en el que las imágenes aparecen elevadas de la superficie de visión; echaremos un vistazo a este efecto aquí. Este realce es más conveniente cuando se trabaja en términos de un *array* de dos dimensiones. Sin embargo, las clases *PixelGrabber* y *MemoryImageSource* sólo funcionan con *arrays* unidimensionales, por lo que, en este ejemplo, simularemos las dos dimensiones multiplicando y sumando los índices del *array*. A continuación, realzamos la imagen *whirl.gif* que vimos en el punto anterior:

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;

/*
```

```
  <APPLET
      CODE=realzar.class
      WIDTH=300
      HEIGHT=300 >
  </APPLET>
*/
public class realzar extends Applet
{
    Image image, image2;

    public void init()
    {
        image = getImage(getDocumentBase(), "whirl1.gif");
        int pixels[] = new int[248 * 248];

      PixelGrabber pg = new PixelGrabber(image, 0, 0, 248, 248, pixels,
0, 248);
        try {
            pg.grabPixels();
        }
        catch (InterruptedException e) {}

        for (int x = 2; x < 247; x++){
           for (int y = 2; y < 247; y++){

               int red = ((pixels[(x + 1) * 248 + y + 1] & 0xFF) -
                    (pixels[x * 248 + y] & 0xFF)) + 128;

          int green = (((pixels[(x + 1) * 248 + y + 1] & 0xFF00) / 0x100)
%
              0x100 - ((pixels[x * 248 + y] & 0xFF00) / 0x100) % 0x100)
+ 128;

             int blue = (((pixels[(x + 1) * 248 + y + 1] &
                  0xFF0000) / 0x10000) % 0x100 - ((pixels[x * 248 + y]
&
                    0xFF0000) / 0x10000) % 0x100) + 128;

               int avg = (red + green + blue) / 3;

               pixels[x * 248 + y] = (0xff000000 | avg << 16 | avg <<
8 | avg);
           }
        }

        image2 = createImage(new MemoryImageSource(248, 248, pixels, 0
, 248));
    }

    public void paint(Graphics g)
    {
        g.drawImage(image2, 0, 0, this);
    }
}
```

El resultado se muestra en la figura 9.16. Como se puede ver, la figura aparece resaltada. Este ejemplo está en el CD como realzar.java.

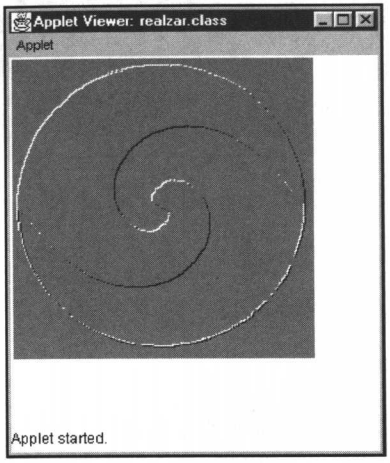


Figura 9.16. Realzar una imagen.

10 AWT: Ventanas, menús y cuadros de diálogo

En este capítulo encontrará el siguiente paso para la construcción de controles en programas AWT. Crearemos y visualizaremos ventanas AWT y veremos todo lo que se va procesando, usando las clases de AWT *Window* y *Frame*. Echaremos un vistazo a cómo se usa la clase *Dialog* para crear cuadros de diálogo y la clase *FileDialog* para crear los cuadros de diálogo de archivos. También revisaremos los menús, ya que en la programación AWT se necesita una ventana antes de poder visualizarlos. Primeros veremos brevemente todos estos elementos.

Ventanas

Las ventanas, por supuesto, son la base de la programación GUI. Todo lo que hay que hacer con la interfaz de usuario en un entorno gráfico tiene lugar en una ventana, y todo usuario GUI está familiarizado con ellas. Para los programadores de AWT hay tres tipos de ventanas.

El primer tipo es la ventana *applet,* en la que la clase *Applet* gestiona la ventana en sí misma y automáticamente la crea y la gestiona. Además, se pueden crear las ventanas *frame*, que son aquellas en las que se piensa

normalmente como ventanas, porque soportan un marco alrededor y una barra de título, así como los botones de maximizar, minimizar y cerrar. Aquí crearemos una, usando la clase *Frame*.



Truco: cuando se lanza una ventana *frame* desde una *applet,* Java da un aviso en el fondo de la ventana por motivos de seguridad: "Aviso: ventana *Applet*", que no es atractivo, pero ahí está.

Hay otro tipo de ventana que se puede utilizar, la clase *Window*. Esta clase sólo presenta una ventana en negro, sin barra de título, sin marco, sólo un rectángulo negro. Cada uno es responsable de visualizar lo que quiera en estas ventanas. Como se podría esperar, la clase *Frame* se deriva de la clase *Window*. Paradójicamente, sin embargo, en los programas no se puede crear, directamente, un objeto de tipo *Window* a menos que ya se tenga una ventana *Frame,* porque el constructor públicamente disponible de *Window* requiere que se le pase un objeto de tipo *Frame* o *Window*.

Menús

Todo usuario GUI conoce los menús, son esos controles indispensables que esconden todas las opciones de un programa y que las presentan por categorías. Imagine si todas las opciones de un procesador de texto que se pueden presentar estuvieran disponibles como botones, visibles todos a la vez; no habría espacio para escribir el texto.

Los menús permiten almacenar esas opciones de forma compacta. Es una técnica GUI muy atractiva, porque el espacio es siempre un grado en los entornos de ventanas.

En la programación AWT, se necesita una ventana *frame* para usar los menús. Con el método de ventana *setMenuBar* se puede crear un objeto *MenuBar* y añadir esa barra de menú a una ventana *frame*. Además se crean objetos de la clase *Menu* para crear menús individuales (como Archivo, Edición, etc.).

Además, se pueden soportar algunas opciones agradables en los menús, submenús que se abren cuando se selecciona un elemento del menú y casillas de activación que le permiten activar o desactivar elementos del menú (como Revisar ortografía automáticamente o Mostrar barra de herramientas). Veremos todo esto en este capítulo.

Cuadros de diálogo

Los programas de ventanas utilizan, con frecuencia, cuadros de diálogo para que el usuario introduzca datos, como el nombre del archivo que se va a abrir, una clave o un color que se selecciona entre muchos otros. Al igual que otros elementos visuales de este capítulo, los cuadros de diálogo son familiares para casi todos los usuarios GUI. Se usan cuando se quiere que el usuario introduzca información, pero no quiere visualizar un control dedicado, como un cuadro de texto, todas las veces en la ventana principal. Es decir, los cuadros de diálogo son ventanas temporales que se pueden rellenar con controles para la entrada de datos del usuario.

En este capítulo, veremos dos tipos de cuadro de diálogo, soportados por las clases *Dialog* y *FileDialog*. Se usa la clase *Dialog* como una clase base para los cuadros de diálogo que se crean y personalizan. Por otro lado, generalmente no se necesita derivar una clase de la clase *FileDialog*, ésta presenta un cuadro de diálogo de archivos que el usuario puede utilizar para seleccionar un archivo. Los métodos y los miembros de datos de esta clase son suficientes para la mayoría de los propósitos de selección de archivos, y lo único que se necesita hacer es instanciar un objeto de esta clase y usarlo. También se verá esto en este capítulo.

Hemos visto rápidamente de qué trata este capítulo. Hay mucho más que ver, por lo que pasaremos a la siguiente sección.

Crear ventanas *Frame*

"De acuerdo", dice el programador novato",estoy trabajando con mi procesador de textos de Java y quiero tener varias vistas del documento, cada una será una ventana situada en un lugar diferente del documento. ¿Qué opina?" Creo que debería pensar en permitir al usuario lanzar nuevas ventanas *frame*", le dice.

Ya se ha visto la clase *Frame* en este libro, ya que se usa como base de la programación de aplicaciones en AWT. Este es el diagrama de herencia de las ventanas *frame:*

```
java.lang.Object
|____java.util.EventObject
     |____java.awt.Component
          |____java.awt.Container
                    |____java.awt.Window
                              |____java.awt.Frame
```

Los constructores y métodos de esta clase se pueden ver en el capítulo 6, en las tablas 6.4 y 6.5, respectivamente. En los siguientes puntos, crearemos una ventana *frame,* que muestre y oculte lo que se necesite, dándole un tamaño, añadiéndole controles y gestionando los eventos. Para empezar, crearemos una nueva clase, *labelFrame,* basada en la clase *Frame,* y se visualizará una etiqueta con el texto "¡Hola desde Java! Esta es una ventana *frame*".

```
Primero, declaramos esta nueva clase extendiendo la clase Frame:
import java.awt.*;

class labelFrame extends Frame
{
   .
   .
   .
}
```

A continuación, crearemos un constructor para la clase. Vamos a utilizar el constructor de la clase *Frame,* que da título a la ventana. Por lo tanto, añadiremos al constructor un parámetro para el título y lo pasaremos a la clase Frame, como sigue:

```
import java.awt.*;

class labelFrame extends Frame
{
   labelFrame(String title)
   {
        super(title);
        .
        .
        .
   }
}
```

Aquí hay un punto importante, el layout manager por defecto para las ventanas *frame* es el gestor *BorderLayout* (a diferencia de las ventanas *applet,* que usan el gestor *FlowLayout*). Observe que esto se aplica tanto a las ventanas *frame* que se visualizan desde las *applets* como a las ventanas *frame* que se usan en las ventanas de aplicación. Si se quiere utilizar un *layout manager* diferente, hay que instalarlo. En este caso, instalaremos el *flow layout manager* usando el método *setLayout*, como sigue:

```
import java.awt.*;

class labelFrame extends Frame
{
   labelFrame(String title)
```

```
    {
        super(title);
        setLayout(new FlowLayout());
        .
        .
        .
    }
}
```

Ahora estamos preparados para crear una etiqueta y añadírsela al esquema:

```
import java.awt.*;

class labelFrame extends Frame
{
  Label label;

  labelFrame(String title)
  {
  super(title);
  setLayout(new FlowLayout());
  label = new Label("¡Hola desde Java! Esta es una ventana frame.");
    add(label);
  }
}
```

Esto completa la clase de ventanas *frame*. El siguiente paso es visualizarla. Ver el siguiente punto para más detalles.

Mostrar y ocultar ventanas

"Hemos creado una clase de ventana *frame*", dice el programador novato. "Ahora, ¿cómo visualizamos una ventana de esa clase?" "No es difícil," le contesta, "después de dar un tamaño a la ventana, sólo hay que usar el método *setVisible*".

Esta es la clase de ventana *frame*, *labelFrame*, que se creó en el punto anterior:

```
import java.awt.*;

class labelFrame extends Frame
{
  Label label;

  labelFrame(String title)
  {
  super(title);
  setLayout(new FlowLayout());
```

```
    label = new Label("¡Hola desde Java! Esta es una ventana frame.");
      add(label);
    }
}
```

Lanzaremos esta ventana desde una *applet*. Para hacer eso, tendremos dos botones: Visualizar la ventana y ocultarla. Así es como se añaden estos botones a la *applet*:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/*
<APPLET
    CODE=frame.class
    WIDTH=300
    HEIGHT=200 >

*/

public class frame extends Applet implements ActionListener {

    Button b1, b2;

    public void init()
    {
        b1 = new Button("Visualizar la ventana");
        add(b1);
        b1.addActionListener(this);

        b2 = new Button("Ocultar la ventana");
        add(b2);
        b2.addActionListener(this);
    .
    .
    .
}
}
```

Ahora, crearemos un nuevo objeto de la clase *labelFrame*, dándole el título "Ventana Java":

```
public class frame extends Applet implements ActionListener {

    Button b1, b2;
    labelFrame window;

    public void init()
    {
        b1 = new Button("Visualizar la ventana");
        add(b1);
```

```
        b1.addActionListener(this);

        b2 = new Button("Ocultar la ventana");
        add(b2);
        b2.addActionListener(this);

        window = new labelFrame("Ventana Java");
        .
        .
        .

}
```

Antes de que se pueda visualizar una ventana, hay que darle una medida, usando el método *setSize*; de lo contrario, no aparecerá en la pantalla. Así es cómo utilizamos *setSize* dándole las dimensiones 300 por 200 pixels:

```
public class frame extends Applet implements ActionListener {

    Button b1, b2;
    labelFrame window;

    public void init()
    {
        b1 = new Button("Visualizar la ventana");
        add(b1);
        b1.addActionListener(this);

        b2 = new Button("Ocultar la ventana");
        add(b2);
        b2.addActionListener(this);

        window = new labelFrame("Ventana Java");
        window.setSize(300, 200);
    }
```

El nuevo objeto ventana ya está preparado. Para visualizar la ventana, se puede usar el método *setVisible*. Se le pasa un valor *True* y para ocultarla, el valor *False*. A continuación se indica el código para estos dos botones en la *applet*:

```
public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == b1){
            window.setVisible(true);
        }
        if(event.getSource() == b2){
            window.setVisible(false);
        }
    }
```

Esto es todo lo que se necesita. La *applet* ya está preparada, como se muestra en la figura 10.1. Cuando se hace clic sobre el botón **Visualizar la**

ventana, la ventana *frame* se muestra, con un luminoso aviso en el fondo de la misma (que no es visible cuando se usa una ventana *frame* en una aplicación), como se puede ver en la figura. Cuando se hace clic sobre el botón **Ocultar la ventana**, ésta desaparece de la pantalla.

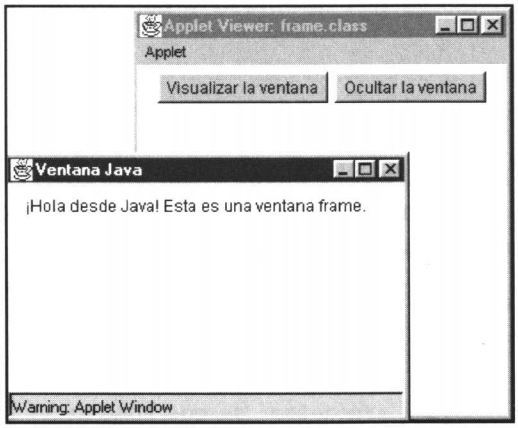


Figura 10.1. Visualizar una ventana *frame*.

Gestionar eventos de ventana

"Hmm", dice el programador novato, "he puesto trescientos botones en una ventana *frame*, pero cuando se hace clic sobre ellos, no pasa nada". Usted sonríe y le dice, "¿Ha gestionado los eventos de hacer clic sobre cada botón en su ventana?" "Uh-oh", dice PN.

Los eventos en las ventanas se pueden gestionar de la misma forma que se gestionan en cualquier sitio, implementando los métodos de las interfaces *listener* o usando las clases adaptador. Vemos un ejemplo en el que gestionamos los eventos de ratón en la ventana *frame* desarrollada en los dos puntos anteriores, implementando los métodos en la interfaz *MouseListener* y luego, visualizando lo que el ratón hace con los mensajes en la etiqueta de esa ventana:

```
class labelFrame extends Frame implements MouseListener {
    Label label;

    labelFrame(String title){
        super(title);
        setLayout(new FlowLayout());
      label = new Label("¡Hola desde Java! Esto es una ventana frame.");
        add(label);
```

```
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent e)
    {
        if((e.getModifiers() & InputEvent.BUTTON1_MASK) ==
            InputEvent.BUTTON1_MASK){
              label.setText("Botón izquierdo del ratón pulsado en "+
e.getX() + "", + e.getY());
        }
        else{
          label.setText("Botón derecho del ratón pulsado en "+ e.getX()
+ "", + e.getY());
        }
    }

    public void mouseClicked(MouseEvent e)
    {
       label.setText("Pulsó el ratón en " + e.getX() + "", + e.getY());
    }

    public void mouseReleased(MouseEvent e)
    {
        label.setText("Se soltó el botón del ratón.");
    }

    public void mouseEntered(MouseEvent e)
    {
        label.setText("Ratón para introducir.");
    }

    public void mouseExited(MouseEvent e)
    {
        label.setText("Ratón para salir.");
    }
}
```

El resultado se muestra en la figura 10.2. Como se puede ver, la ventana *frame* gestiona eventos *MouseListener,* visualizando mensajes. Este programa se encuentra en el CD que acompaña a este libro, como *frame.java.*

Además hay eventos específicos para ventanas, es decir, hay toda una interfaz *WindowListener.* Sus métodos se pueden encontrar en la tabla 10.1.

Tabla 10.1. Métodos de la interfaz *WindowListener.*

Métodos	Descripción
void windowActivated(WindowEvent e)	Se le llama cuando la ventana es la ventana activa del usuario, lo que quiere decir que la ventana recibirá los eventos de teclado.

Métodos	Descripción
void windowClosed(WindowEvent e)	Se le llama cuando una ventana se cierra.
void windowClosing(WindowEvent e)	Se le llama cuando el usuario cierra la ventana desde el menú del sistema de la ventana.
void windowDeactivated(windowEvent e)	Se le llama cuando una ventana no es más grande que la ventana activa del usuario.
void windowDeiconified(WindowEvent e)	Se le llama cuando una ventana se cambia de un estado minimizado a uno normal.
void windowIconified(WindowEvent e)	Se le llama cuando una ventana se cambia desde un estado normal a uno minimizado.
void windowOpened(WindowEvent e)	Se le llama la primera vez que una ventana se hace visible.

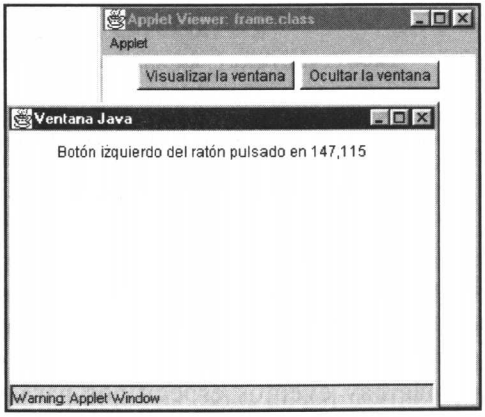


Figura 10.2. Gestionar eventos de ratón en una ventana *frame*.

Observe que la clase *WindowAdapter* se puede usar también para gestionar los eventos ventana. Esta clase ya implementa todos los métodos de la interfaz *WindowListener* y sólo se sobrescriben los métodos que se quieran usar. A los métodos de la interfaz *WindowListener* y de la clase *WindowAdapter* se les pasa un objeto de la clase *WindowEvent*. Los campos de esta clase se pueden ver en la tabla 10.2, su constructor en la 10.3 y sus métodos en la 10.4.

En el siguiente punto, veremos uno de los métodos *WindowInterface* trabajando junto con el evento *windowClosing*.

Tabla 10.2. Campos de la clase *WindowEvent*.

Campo	Descripción
static int WINDOW_ACTIVATED	Evento "ventana activada".
static int WINDOW_CLOSED	Evento "ventana cerrada".
static int WINDOW_CLOSING	Evento "ventana cerrándose".
static int WINDOW_DEACTIVATED	Evento "ventana desactivada".
static int WINDOW_DEICONFIED	Evento "ventana sin iconos".
static int WINDOW_FIRST	Primer número de ID dentro del rango usado para los eventos de ventana.
static int WINDOW_ICONFIED	Evento "ventana iconificada".
static int WINDOW_LAST	Último número de ID dentro del rango usado para los eventos de ventana.
static int WINDOW_OPENED	Evento "ventana abierta".

Tabla 10.3. Constructor de la clase *WindowEvent*.

Constructor	Descripción
WindowEvent(Window source, int id)	Crea un objeto *WindowEvent*.

Tabla 10.4. Métodos de la clase *WindowEvent*.

Método	Descripción
Window getWindow()	Obtiene el originador del evento.
String paramString()	Obtiene una cadena que identifique el evento.

Ocultar ventanas automáticamente al cerrarlas

"Hey", dice el programador novato, "otra vez Java está graciosillo. Cuando intento cerrar una ventana*frame*, no hace nada. Voy a enviar un correo a Sun Microsystems sobre esto, "En realidad", le dice, "está cerrando una ventana por usted, eso es todo". "Oh, " dice PN, "¿cómo funciona eso?"

Para cerrar una ventana *frame* cuando el usuario hace clic sobre el botón **Cerrar** en la esquina superior derecha de la barra de título, se puede gestionar el evento *windowClosing*. Aquí, estamos usando una clase interna adaptador anónima para hacerlo (ver el capítulo 6 para más detalles). En este caso, sólo se esconde la ventana cuando el usuario hace clic sobre el botón **Cerrar**:

```
class labelFrame extends Frame
{
  Label label;

  labelFrame(String title)
  {
  super(title);
  setLayout(new FlowLayout());
  label = new Label("¡Hola desde Java! Esta es una ventana  frame.");
    add(label);
  addMouseListener(this);
 addWindowListener WindowAdapter() {public void windowClosing(WindowEvent
e)  {setVisible(false);}});
  }
}
```

Hay algo útil que hacer notar. Cuando se cierra una ventana de esta forma, todavía existe un objeto, lo que significa que todavía se puede acceder a sus métodos y miembros de datos. Esto será útil cuando vayamos a trabajar con cuadros de diálogo y se quieran recuperar los datos que el usuario metió.

Usar la clase *Window*

"Quiero personalizar la apariencia de mi ventana totalmente", dice el programador novato. "Voy a visualizar los trabajos de los grandes maestros de pintura y quiero hacer que el borde parezca un marco de un cuadro real". "Hmm", le dice, "suena muy bien. Debería usar la clase *Window* y personalizarla".

Para crear una ventana en negro preparada para personalizarse, se puede usar la clase *Window*. Los constructores de esta clase se encuentran en la tabla 10.5 y sus métodos en la 10.6.

Tabla 10.5. Constructores de la clase *Window*.

Constructor	Descripción
*window(Frame*propietario*)*	Crea una nueva ventana invisible.
*Window(Window*propietario*)*	Crea una ventana nueva invisible con la ventana dada como su dueño.

Tabla 10.6. Métodos de la clase *Window*.

Método	Descripción
void addNotify()	Hace que la ventana sea visible, conectando la ventana al sistema operativo.
void addWindowListener (WindowListener l)	Añade el *Window listener* dado para obtener los eventos de ventana desde la misma.
void applyResourceBundle (ResourceBundle rb)	Aplica las propiedades del recurso dado a la ventana.
void applyResourceBundle (String rbName)	Carga el recurso con el nombre dado.
void dispose()	Libera todos los recursos de pantalla nativos usados por esta ventana.
protected void finalize()	Finaliza los métodos de entrada y contexto.
Component getFocusOwner()	Obtiene el componente hijo de esta ventana, que tiene el foco si y sólo si la ventana está activa.
InputContext getInputContext()	Obtiene el contexto de entrada para esta ventana.
Locate getLocale()	Obtiene el objeto *locale* asociado a esta ventana si el *locale* está puesto.
Window[] getOwnedWindows()	Devuelve un *array* que contiene todas las ventanas a las que esta ventana pertenece realmente.
Window getOwner()	Obtiene el propietario de esta ventana.
Toolkit gettoolkit()	Obtiene las herramientas del *frame*.
String getWarningString()	Obtiene la cadena de aviso visualizada con esta ventana.
void hide()	Esconde esta ventana.
boolean isShowing()	Verifica si esta ventana es visible.
void pack()	Hace que esta ventana sea medida para encajar la medida preferida y los esquemas de sus componentes.
boolean postEvent(Event e)	Obsoleto. Reemplazado por *dispatchEvent-(AWTEvent)*.
protected void processEvent (AWTEvent e)	Procesa eventos en esta ventana.

Método	Descripción
protected void processWindowEvent(WindowEvent e)	Procesa los eventos de ventana que ocurren en la misma, enviándolos a todos los objetos *WindowListener* registrados.
void removewindowListener (WindowListener l)	Elimina el *window listener* dado.
void setCursor(Cursor cursor)	Establece que la apariencia del cursor sea la del cursor especificado.
void show()	Hace que la ventana sea visible.
void toBack()	Envía esta ventana a la parte de atrás.
void toFront()	Trae la ventana al frente.

Veamos un ejemplo. En este caso, creamos una nueva clase de ventana llamada *labelWindow* que visualiza una etiqueta con el texto "¡Hola desde Java!". Empecemos por declarar esta nueva clase:

```
class labelWindow extends Window
{
   .
   .
   .
}
```

Tenemos que pasar un objeto *Frame* u otro objeto *Window* al constructor de la clase *Window*. Para hacer las cosas más sencillas, usamos esta nueva ventana en una aplicación, por lo que pasaremos el objeto principal de la ventana *frame* al constructor de la clase *labelWindow* y luego ese objeto se le pasa al constructor de la clase *Window*, como sigue:

```
class labelWindow extends Window {
    Label label;

    labelWindow(AppFrame af){
        super(af);
   .
   .
   .
     }
}
```

Al igual que con las ventanas *frame*, el gestor de esquemas por defecto es un *border layout*, por lo que instalaremos un *flow layout* y visualizaremos la etiqueta con su mensaje en la ventana:

```
class labelWindow extends Window {
    Label label;

    labelWindow(AppFrame af){
        super(af);
        setLayout(new FlowLayout());
        label = new Label("¡Hola desde Java!");
        add(label);
    }
    .
    .
    .
}
```

Las ventanas de la clase *Window* son sencillamente negras con espacios blancos, por lo que además sobrescribiremos el método *paint* de esta ventana para añadir un *frame* muy sencillo, como un rectángulo. Observe que se pueden determinar las dimensiones de la ventana con el método *getSize*, al igual que en la *applet*:

```
class labelWindow extends Window {
    Label label;

    labelWindow(AppFrame af){
        super(af);
        setLayout(new FlowLayout());
        label = new Label("¡Hola desde Java!");
        add(label);
    }

    public void paint (Graphics g)
    {
        int width = getSize().width;
        int height = getSize().height;

        g.drawRect(0, 0, -width, -height);
    }
}
```

Esto completa la clase *labelWindow*. Ahora, pondremos en funcionamiento esta clase en una aplicación. En este caso, instanciaremos un objeto de la clase y fijaremos su tamaño, esto es necesario antes de que la ventana se pueda visualizar. Además pondremos su ubicación en la pantalla usando *setLocation*, porque actualmente no hay forma de mover la ventana cuando aparece, y si sólo se muestra con el método *setVisible*, aparecerá en la posición (0,0) de la pantalla, que está en el extremo de la esquina superior izquierda. Cuando el usuario hace clic en el botón **Visualizar la ventana** de esta aplicación, la ventana aparecerá. Cuando el usuario hace clic sobre el botón **Esconder la ventana**, ésta desaparecerá. Este es el código:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class window
{
    public static void main(String [] args)
    {
       AppFrame f = new AppFrame();

       f.setSize(200, 200);

       f.addWindowListener(new WindowAdapter() { public void
           windowClosing(WindowEvent e) {System.exit(0);}});

       f.show();
    }
}

class AppFrame extends Frame implements ActionListener
{
    Button b1, b2;
    labelWindow window;

    AppFrame()
    {
        setLayout(new FlowLayout());
        b1 = new Button("Visualizar la ventana");
        add(b1);
        b1.addActionListener(this);

        b2 = new Button("Ocultar la ventana");
        add(b2);
        b2.addActionListener(this);

        window = new labelWindow(this);
        window.setSize(300, 200);
        window.setLocation(300, 300);
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == b1){
            window.setVisible(true);
        }
        if(event.getSource() == b2){
            window.setVisible(false);
        }
    }
}
```

El resultado se muestra en la figura 10.3. Como se puede ver, la ventana aparece con un borde mínimo y la etiqueta de control que visualiza un mensaje. Esta aplicación se puede encontrar en el CD como ventana.java.

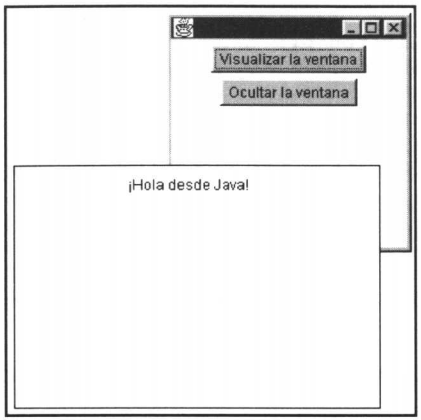


Figura 10.3 Crear una ventana de la clase *Window*.

Crear menús

"De acuerdo", dice el programador novato, "he estado esperando para preguntarle esto". "Ah, ¿sí?", le dice. El programador novato pregunta, "¿cómo se crean los menús?" Se sienta y le dice, "Mejor tomemos un café".

Se pueden añadir menús AWT a ventanas de la clase *Frame* usando tres clases AWT: *MenuBar, Menu* y *MenuItem*. Aquí veremos los detalles de las tres. La primera que utilizaremos es la clase *MenuBar*, que añade una barra de menú a una ventana *frame*. Después de que se ha añadido una barra de menú a una ventana *frame*, se pueden añadir menús, como el menú Archivo o el menú Edición, a esa barra de menú usando la clase *Menu*. Finalmente, en el programa añadimos los elementos a los menús, usando la clase *MenuItem*.

Crearemos y usaremos menús en los siguientes puntos mientras desarrollamos una *applet* llamada menu.java, que encontrará en el CD. Esta *applet* visualizará una ventana *frame* con un menú. Esta es la *applet:*

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/*
<APPLET
    CODE=menu.class
    WIDTH=200
    HEIGHT=200 >

*/
<APPLET
public class menu extends Applet implements ActionListener
```

```
{
    Button b1;
    frame menuWindow;

    public void init()
    {
        b1 = new Button("Visualizar el menú de la ventana");
        add(b1);
        b1.addActionListener(this);

        menuWindow = new frame("Menús");
        menuWindow.setSize(200, 200);
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == b1){
            menuWindow.setVisible(true);
        }
    }
}

class frame extends Frame implements ActionListener
{
    Menu menu;
    MenuBar menubar;
    MenuItem menuitem1, menuitem2, menuitem3;

    Label label;

    frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();

        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
        menu.add(menuitem1);
        menuitem1.addActionListener(this);

        menuitem2 = new MenuItem("Elemento 2");
        menu.add(menuitem2);
        menuitem2.addActionListener(this);

        menuitem3 = new MenuItem("Elemento 3");
        menu.add(menuitem3);
        menuitem3.addActionListener(this);

        menubar.add(menu);

        setMenuBar(menubar);
```

```
        addWindowListener(new WindowAdapter() {public void
            windowClosing(WindowEvent e) {setVisible(false);}});
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == menuitem1){
            label.setText("Eligió el elemento 1");
        } else if(event.getSource() == menuitem2){
            label.setText("Eligió el elemento 2");
        } else if(event.getSource() == menuitem3){
            label.setText("Eligió el elemento 3");
        }
    }
}
```

En la figura 10.4 se muestra esta *applet* en funcionamiento.

El menú de esta *applet* ocupa un lugar en la clase de la ventana *Frame* que hemos llamado *frame*. Desarrollaremos esta clase por medio de ejemplos a lo largo de los siguientes puntos.

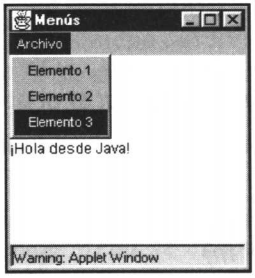


Figura 10.4. Crear una ventana de la clase *Window*.

Crear un objeto *MenuBar*

"Estoy preparado para crear un menú de sistema en mi ventana *frame*", dice el programador novato. "¿Por dónde empiezo?" "Empiece creando un objeto *MenuBar*. Siéntese y le explicaré cómo", le contesta.

Para crear una barra de menús en una ventana *frame*, se usa la clase *MenuBar*. Este es el diagrama de herencia de la clase *MenuBar*:

```
java.lang.Object
|____java.awt.MenuComponent
     |____java.awt.MenuBar
```

El constructor de esta clase se encuentra en la tabla 10.7 y sus métodos en la 10.8.

Tabla 10.7. Constructor de la clase *MenuBar*.

Constructor	Descripción
MenuBar()	Crea una nueva barra de menús.

Tabla 10.8. Métodos de la clase *MenuBar*.

Método	Descripción
Menu add(Menu m)	Añade el menú dado a la barra de menú.
void addNotify()	Crea el compañero de la barra de menú.
int countMenus()	Obsoleto. Reemplazado por *getMenuCount()*.
void deleteShortCut (MenuShortCut s)	Borra el menú dado.
Menu getHelpMenu()	Obtiene el menú de ayuda de la barra de menú.
Menu getMenu(int I)	Obtiene el menú dado.
int getMenuCount()	Obtiene el número de menús de la barra de menús.
MenuItem getShortcut-MenuItem(MenuShortcut s)	Obtiene la instancia de *MenuItem* asociada con el objeto *MenuShortcut* dado.
void remove(int indice*)*	Elimina el menú localizado en el índice dado de esta barra de menús.
void remove(Menu-Component m)	Elimina el componente dado de esta barra de menú.
void removeNotify()	Elimina el compañero de la barra de menú.
void setHelpMenu (Menu m)	Establece que el menú de ayuda del menú dado es el de esta barra de menú.
Enumeration shortcuts()	Obtiene una enumeración de todas las formas abreviadas para usar el teclado de los menús.

A continuación encontrará cómo creamos una barra de menú en la ventana *frame* de la *applet* menu.java, construida en el punto "Crear menús" (observe que instalamos la barra de menú en la ventana *frame* con el método *setMenuBar* de la clase *Frame*:

```
class frame extends Frame implements ActionListener
{
    MenuBar menubar;
    Label label;

    frame(String title)
```

```
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();
        .
        .
        .
        setMenuBar(menubar);
    }
}
```

Con esto, se visualiza una barra de menús en negro y es el momento de añadir algunos menús. Para más detalles, ver el siguiente apartado.

Crear objetos *Menu*

"Bien", dice el programador novato, "me he dado cuenta de que después de crear una nueva barra de menús, está vacía. ¿Cómo puedo añadir menús?" "Es fácil", le dice, "sólo hay que usar la clase *Menu*".

Con la clase *Menu,* se crean los menús individuales de la barra de menús. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____java.awt.MenuComponent
     |____java.awt.MenuItem
                    |____java.awt.Menu
```

Los constructores de la clase *Menu* se encuentran en la tabla 10.9 y sus métodos en la tabla 10.10.

Tabla 10.9. Constructores de la clase *Menu.*

Constructor	Método
Menu()	Crea un menú nuevo.
Menu(String etiqueta*)*	Crea un menú nuevo con la etiqueta dada.
Menu(String etiqueta, *boolean tearOff)*	Crea un nuevo menú con la etiqueta dada, indicando si el menú puede partirse.

Tabla 10.10. Métodos de la clase *Menu.*

Método	Descripción
MenuItem add(MenuItem m)	Añade el elemento de menú dado.

Método	Descripción
void add(String etiqueta)	Añade un elemento a este menú, con la etiqueta dada.
void addNotify()	Crea el compañero del menú.
void addSeparator()	Añade una línea o un guión de separación al menú, en la posición actual.
int countItems()	Obsoleto. Reemplazado por *getI temCount.*
MenuItem getItem(int indice)	Obtiene el elemento localizado en el índice dado.
int getItemCount()	Obtiene el número de elementos.
void insert(MenuItem menuitem, int indice)	Inserta, en la posición dada, un elemento del menú.
void insert(String etiqueta, *int* indice)	Inserta un elemento de menú con la etiqueta dada.
*void insertSeparator(int*indice)	Inserta un separador en la posición dada.
boolean isTearOff()	Indica si este menú es un menú partido.
String paramString()	Obtiene la cadena que representa el estado de este menú.
void remove(int indice)	Elimina el elemento del menú que se encuentra en la posición dada.
void removeAll()	Elimina todos los elementos de este menú.
void removeNotify()	Elimina el compañero del menú.

A continuación, se indica cómo se crea un menú Archivo en la *applet* menu.java y se añade a la barra de menú (observe que se nombra al menú simplemente pasando el nombre al constructor de la clase *Menu*):

```
class frame extends Frame implements ActionListener
{
    Menu menu;
    MenuBar menubar;
    Label label;
```

```
    frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();

        menu = new Menu("Archivo");

            .
            .
            .

        menubar.add(menu);

        setMenuBar(menubar);
    }
}
```



Truco: una buena forma potente de crear menús es como si estuvieran partidos, lo cual se puede lograr utilizando la tercera forma del constructor *Menu,* que se encuentra en la tabla 10.9. Los usuarios pueden partir estos menús con el ratón y situarlos donde quieran.

Crear objetos *MenuItem*

El programador novato regresa y dice, "Hey, he creado un nuevo menú. ¿Cómo puedo añadir elementos a ese menú?" "No es tan difícil", le dice, "basta con usar la clase *MenuItem*".

Cada elemento de menú que se añade a un menú AWT es realmente un objeto de la clase *MenuItem*. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____java.awt.MenuComponent
     |____java.awt.MenuItem
```

Los constructores de esta clase se encuentran en la tabla 10.11 y sus métodos en la 10.12.

Tabla 10.11. Constructores de la clase *MenuItem.*

Constructor	Descripción
MenuItem()	Crea un nuevo elemento de menú.

Constructor	Descripción
MenuItem(String etiqueta)	Crea un nuevo elemento con la etiqueta dada y sin posibilidad de activarlo de forma abreviada con el teclado.
Menuitem(String etiqueta, *MenuShortcut s)*	Crea un nuevo elemento con una forma abreviada para activarlo con el teclado.

Tabla 10.12. Métodos de la clase *MenuItem.*

Método	Descripción
void addActionListener(ActionListener l)	Añade el *action listener*dado para recibir eventos de acción desde este elemento de menú.
void addNotify()	Crea el compañero del elemento de menú.
void deleteShortcut()	Borra cualquier objeto*MenuShortcut* asociado con este elemento de menú.
void disable()	Obsoleto. Reemplazado por *setEnabled(boolean).*
protected void disableEvents(long eventsToDisable)	Deshabilita la entrega de eventos a este elemento de menú.
void enable()	Obsoleto. Reemplazado por *setEnabled(boolean).*
void enable(boolean b)	Obsoleto. Reemplazado por *setEnabled(boolean).*
protected void enableEvents(long eventsToEnable)	Habilita la entrega de eventos a este elemento de menú.
String getActionCommand()	Obtiene el nombre del comando del evento de acción que ha sido producido por este elemento de menú.
String getLabel()	Obtiene la etiqueta de este elemento de menú.
MenuShortcut getShortcut()	Obtiene el objeto *MenuShortcut* asociado con este elemento de menú.
boolean isEnabled()	Verifica si este elemento de menú está habilitado.

Método	Descripción
String paramString()	Obtiene la cadena de parámetros que representa el estado de este elemento de menú.
protected void processEvent (AWTEvent e)	Procesa eventos en este elemento de menú.
void removeActionListener (ActionListener l)	Elimina el *action listener* dado, por lo que no obtiene más eventos de acción de este elemento.
void setActionCommand(String comando)	Fija el nombre del comando del evento de acción que está activado por el elemento del menú.
void setEnabled(boolean b)	Fija si se puede elegir este elemento de menú.
void setLabel(String etiqueta)	La etiqueta de este elemento es la etiqueta dada.
void setShortcut(MenuShortcut s)	Fija el objeto *MenuShortcut* asociado con este elemento.

Para añadir un elemento de menú a un menú, basta con crear un nuevo objeto *MenuItem*, pasando al constructor *MenuItem* el nombre del elemento nuevo y después usando el método *add* del objeto *Menu* para añadirlo a un menú. Este es un ejemplo en el que se añaden elementos al menú Archivo de la *applet* menu.java:

```
class frame extends Frame implements ActionListener
{
    Menu menu;
    MenuBar menubar;
    MenuItem menuitem1, menuitem2, menuitem3;

    Label label;

    frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();

        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
        menu.add(menuitem1);
```

```
        menuitem1.addActionListener(this);

        menuitem2 = new MenuItem("Elemento 2");
        menu.add(menuitem2);
        menuitem2.addActionListener(this);

        menuitem3 = new MenuItem("Elemento 3");
        menu.add(menuitem3);
        menuitem3.addActionListener(this);

        menubar.add(menu);

        setMenuBar(menubar);
    }
}
```



Truco: Java permite que un elemento de menú pueda ser activado utilizando una combinación de teclas. Esto se puede hacer usando la tercera forma del constructor *MenuItem* como se indica en la tabla 10.11. Los usuarios pueden acceder a estos elementos con sólo pulsar las teclas asociadas.

Ahora lo que ocurre es que cuando estos elementos son pulsados no hacen nada. Para aprender a soportar los eventos de menús, ver el siguiente punto.

Gestionar los eventos de menú

El programador novato (PN) regresa y dice, "Bien, ya tengo todo el menú del sistema, pero hay un problema. Cuando se hace clic sobre los elementos del menú no pasa nada". "Eso es porque hay que añadir un *action listener* a cada uno de ellos", le dice riendo. "Oh", dice PN.

Los *action listeners* para los elementos de menú se utilizan igual que con los botones. Este es un ejemplo en el que se habilita la gestión de eventos de menú en la *applet* menu.java.

En este caso, se indica qué elemento ha seleccionado el usuario, visualizando un mensaje en una etiqueta. Primero, se añade un *action listener* a cada uno de los elementos:

```
class frame extends Frame implements ActionListener
{
    Menu menu;
    MenuBar menubar;
    MenuItem menuitem1, menuitem2, menuitem3;
```

```
    Label label;

    frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();

        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
        menu.add(menuitem1);
        menuitem1.addActionListener(this);

        menuitem2 = new MenuItem("Elemento 2");
        menu.add(menuitem2);
        menuitem2.addActionListener(this);

        menuitem3 = new MenuItem("Elemento 3");
        menu.add(menuitem3);
        menuitem3.addActionListener(this);

        menubar.add(menu);

        setMenuBar(menubar);

         addWindowListener(new WindowAdapter() {public void
              windowClosing(WindowEvent e) {setVisible(false);}});
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == menuitem1){
            label.setText("Eligió el elemento 1");
        } else if(event.getSource() == menuitem2){
            label.setText("Eligió el elemento 2");
        } else if(event.getSource() == menuitem3){
            label.setText("Eligió el elemento 3");
         }
    }
}
```

Ahora, el usuario puede ejecutar la *applet* menu.java y verá los elementos que seleccionó, como se muestra en la figura 10.5.

Aunque utilizamos el método *getSource* de la clase *ActionEvent* para determinar qué elemento de menú fue pulsado, también se puede dar a cada elemento un comando de acción con el método *setActionCommand,* como se hizo para los botones en el capítulo 6, y usar el método *getActionCommand* de la clase *ActionEvent* para leer el comando de acción.

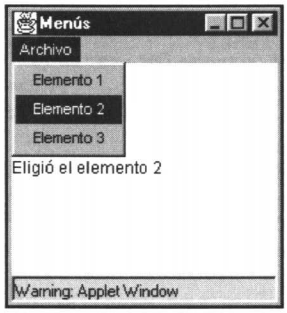


Figura 10.5. Crear y usar un menú del sistema.

Más opciones de menú

En los puntos anteriores, creamos una *applet,* menu.java, que examinaba la construcción básica de un menú de sistema. En los siguientes apartados, lo elaboraremos más e incluiremos separadores de menú, elementos deshabilitados, casillas de activación con elementos de menús y submenús, todo en una nueva *applet* llamada menu2.java.

Así es la nueva *applet:*

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/*
<APPLET
    CODE=menu2.class
    WIDTH=200
    HEIGHT=200 >

*/
<APPLET
public class menu2 extends Applet implements ActionListener
{
    Button b1;
    frame menuWindow;

    public void init()
    {
        b1 = new Button("Visualizar el menú de la ventana");
        add(b1);
        b1.addActionListener(this);

        menuWindow = new frame("Menús");
        menuWindow.setSize(200, 200);
    }

    public void actionPerformed(ActionEvent event)
```

```
        {
            if(event.getSource() == b1){
                menuWindow.setVisible(true);
            }
        }
    }

    class frame extends Frame implements ActionListener
    {
        Menu menu, submenu;
        MenuBar menubar;
        MenuItem menuitem1, menuitem2, menuitem4;
        MenuItem subitem1, subitem2, subitem3;
        CheckboxMenuItem menuitem3;
        Label label;

        frame(String title)
        {
            super(title);
            label = new Label("¡Hola desde Java!");
            setLayout(new GridLayout(1, 1));
            add(label);
            menubar = new MenuBar();
            menu = new Menu("Archivo");

            menuitem1 = new MenuItem("Elemento 1");
            menu.add(menuitem1);
            menuitem1.addActionListener(this);

            menuitem2 = new MenuItem("Elemento 2");
            menu.add(menuitem2);
            menuitem2.addActionListener(this);

            menu.addSeparator();

               menuitem3 = new CheckboxMenuItem("Elemento con casilla de
activación");
            menuitem3.addActionListener(this);
            menu.add(menuitem3);

            menu.addSeparator();

            submenu = new Menu("Sub menús");
            subitem1 = new MenuItem("Sub elemento 1");
            subitem2 = new MenuItem("Sub elemento 2");
            subitem3 = new MenuItem("Sub elemento 3");
            subitem1.addActionListener(this);
            subitem2.addActionListener(this);
            subitem3.addActionListener(this);
            menuitem2.addActionListener(this);
            menuitem2.addActionListener(this);
            submenu.add(subitem1);
            submenu.add(subitem2);
            submenu.add(subitem3);
```

```
        menu.add(submenu);

        menu.addSeparator();

        menuitem4 = new MenuItem("Salir");
        menuitem4.addActionListener(this);
        menu.add(menuitem4);

        menubar.add(menu);
        setMenuBar(menubar);

        addWindowListener(new WindowAdapter() {public void
            windowClosing(WindowEvent e) {setVisible(false);}});
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == menuitem1){
            label.setText("Eligió el elemento 1");
        } else if(event.getSource() == menuitem2){
            label.setText("Eligió el elemento 2");
        } else if(event.getSource() == menuitem3){
            label.setText("Eligió el elemento 3");
        }
    }

    public void itemStateChanged (ItemEvent event)
    {
          if(event.getSource() == menuitem3){
      if(((CheckboxMenuItem)event.getItemSelectable()).getState())
                    label.setText("El elemento 3 está selecciona-
do");
                else
                    label.setText("El elemento 3 no está seleccio-
nado");
                }
    }
}
```

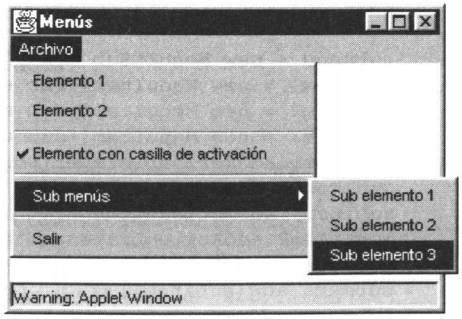


Figura 10.6. Elaborar un menú de sistema.

Esta *applet* se puede ver en la figura 10.6. Desarrollaremos varios nuevos aspectos de esta *applet,* menu2.java, en los siguientes puntos, cuando veamos cómo se gestionan varias partes de su código.

Añadir separadores de menú

Los elementos de menú se pueden agrupar usando los separadores de menú, líneas finas horizontales que aparecen en los menús (ver la figura 10.6).

Añadir un separador de menú es fácil, sólo hay que usar el método *addSeparator*, como sigue:

```
frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();
        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
        menuitem1.addActionListener(this);
        menu.add(menuitem1);

        menuitem2 = new MenuItem("Elemento 2");
        menuitem2.addActionListener(this);
        menu.add(menuitem2);

        menu.addSeparator();
        .
        .
        .
    }
```

Deshabilitar elementos de menú

"Este maldito Johnson", dice el programador novato, "ya ha estado ensuciando mi programa otra vez, seleccionando elementos de menú que no debería, como la revisión ortográfica cuando el programa dibuja gráficos. ¡Hizo que explotara mi programa delante del gran jefe!" "Hmm", le dice, "¿qué tal si se deshabilitan los elementos del menú cuando no son adecuados?" PN sonríe y dice, "¡Va a ver ese Johnson!"

Cuando se deshabilita un elemento de menú, se visualiza en gris y no se puede seleccionar ni hacer clic sobre él. Para deshabilitar un elemento de menú, se usa el método *setEnabled* de ese elemento, pasándole un valor de *False*.

Por ejemplo, se puede deshabilitar el segundo elemento de menú en menu2.java cuando el usuario haga clic sobre él, como sigue:

```
class frame extends Frame implements ActionListener
{
    Menu menu, submenu;
    MenuBar menubar;
    Label label;
    MenuItem menuitem1, menuitem2, menuitem4;
    MenuItem subitem1, subitem2, subitem3;
    CheckboxMenuItem menuitem3;

    frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();
        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
        menuitem1.addActionListener(this);
        menu.add(menuitem1);

        menuitem2 = new MenuItem("Elemento 2");
        menuitem2.addActionListener(this);
        menu.add(menuitem2);

        menubar.add(menu);
        setMenuBar(menubar);
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == menuitem1){
            label.setText("Eligió el elemento 1");
        } else if(event.getSource() == menuitem2){
            menuitem2.setEnabled(false);
            label.setText("Eligió el elemento 2");
        }
    }
}
```

El resultado se muestra en la figura 10.7, donde se puede ver el elemento deshabilitado.

Ahora que el elemento está deshabilitado, el usuario no tiene forma de hacer clic sobre él hasta que no esté habilitado de nuevo.

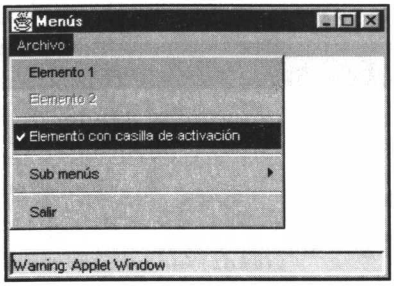


Figura 10.7. Deshabilitar un elemento de menú.

Añadir marcas de activación a menús

El especialista en soporte de productos regresa descontento. "Hay un problema en su nueva aplicación", dice, "porque el usuario no tiene forma de saber si la opción de menú "Traducción automática a alemán" está activada o no, y los resultados son bastante molestos". "Hmm", le dice, "supongo que puedo añadir una marca de activación a esa opción de menú para mostrar cuándo tiene efecto".

Se pueden crear elementos de menú que visualizan marcas de activación junto a ellos para indicar que esas opciones específicas tienen efecto. Además se puede mantener hundida esa marca cuando el usuario la selecciona. Para soportar este tipo de marcas, usamos la clase *CheckboxMenuItem,* que tiene el siguiente diagrama de herencia:

```
java.lang.Object
     |____java.awt.MenuComponent
          |____java.awt.CheckboxMenuItem
```

Los constructores de *CheckboxMenuItem* se encuentran en la tabla 10.13 y sus métodos en la 10.14.

Tabla 10.13. Constructores de la clase *CheckboxMenuItem.*

Constructor	Descripción
CheckbosMenuItem()	Crea un elemento con casilla de activación.
*CheckboxMenuItem(String*etiqueta)	Crea un elemento con casilla de activación con la etiqueta dada.
*CheckboxMenuItem(String*etiqueta, *boolean* estado)	Crea un elemento con casilla de activación con la etiqueta y el estado dados.

Tabla 10.14. Métodos de la clase *CheckboxMenuItem.*

Método	Descripción
void additemListener(ItemListener l)	Añade el *item listener* dado para recibir los eventos de este elemento.
void addNotify()	Crea el compañero de ese elemento.
Object[] getSelectedObjects()	Obtienen un *array* de longitud 1 que contiene la etiqueta del elemento.
boolean getState()	Determina si el estado de este elemento es activado o no.
String paramString()	Obtiene una cadena que representa el estado de este elemento.
protected void processEvent (AWTEvent e)	Procesa eventos en este elemento.
protected void processEvent (ItemEvent e)	Procesa los eventos que ocurren en este elemento enviándolos a objetos *ItemListener.*
void removeItemListener(item-Listener l)	Elimina el *item listener* dado para que no reciba más eventos de este elemento.
void setState(boolean b)	Fija el elemento al estado dado.

Observe que usamos *item listeners,* no *action listeners,* con este tipo de elementos de menús. Aquí vemos cómo se añade un elemento con casilla de activación en la *applet* menu2.java, usando un objeto *ItemListener:*

```
class frame extends Frame implements ActionListener, ItemListener{
    Menu menu, submenu;
    MenuBar menubar;
    Label label;
    MenuItem menuitem1, menuitem2, menuitem4;
    MenuItem subitem1, subitem2, subitem3;
    CheckboxMenuItem menuitem3;

    frame(String title)
    {
        super(title);
        label = new Label(";Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();
        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
```

```
        menuitem1.addActionListener(this);
        menu.add(menuitem1);

        menuitem2 = new MenuItem("Elemento 2");
        menuitem2.addActionListener(this);
        menu.add(menuitem2);

        menu.addSeparator();

       menuitem2 = new MenuItem("Elemento con casilla de activación");
        menuitem3.addItemListener(this);
        menu.add(menuitem3);
         .
         .
         .
        menubar.add(menu);
        setMenuBar(menubar);

         addWindowListener(new WindowAdapter() {public void
              windowClosing(WindowEvent e) {setVisible(false);}});
    }
```

Cuando el usuario selecciona este elemento una vez tras otra, Java hunde la marca de activación o no, automáticamente. Para gestionar los eventos de este elemento, hay que sobrescribir el método *itemStateChanged*. En este caso, determinamos el estado del elemento y lo visualizamos en una etiqueta:

```
public void itemStateChanged (ItemEvent event)
    {
  if(event.getSource() == menuitem3){
if(((CheckboxMenuItem)event.getItemSelectable()).getState())
                label.setText("El elemento 3 está seleccionado");
            else
                label.setText("El elemento 3 no está seleccionado");
            }
    }
}
```

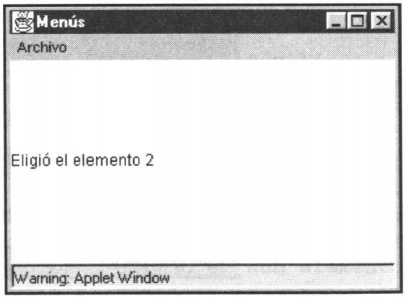


Figura 10.8. Usar un elemento con casilla de activación.

En la figura 10.8 se puede ver un elemento de este tipo funcionando.

Crear submenús

Otro aspecto importante de trabajar con menús en la programación de AWT involucra a los submenús. Están unidos a un menú y cuando el usuario selecciona el elemento, el submenú se abre, como se puede ver en la figura 10.9. El usuario puede seleccionar los elementos del submenú de la misma forma que lo hace en los menús normales.

Usar la técnica de los submenús es muy interesante cuando se necesita otro nivel de detalle; por ejemplo, quizás se quiera permitir que el usuario seleccione un color de dibujo con un elemento de menú, y cuando esté seleccionado, que un submenú se abra indicando los colores posibles, rojo, verde, magenta y azul.

Crear un submenú es fácil, sólo hay que añadir elementos de menú a otro elemento usando el método *add*. Este es un ejemplo que muestra cómo funciona. En este caso, se añaden los elementos de submenú que se ven en la figura 10.9 a los elementos de los submenú. Así sería el código:

```
frame(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        menubar = new MenuBar();
        menu = new Menu("Archivo");

        menuitem1 = new MenuItem("Elemento 1");
        menuitem1.addActionListener(this);
        menu.add(menuitem1);
```

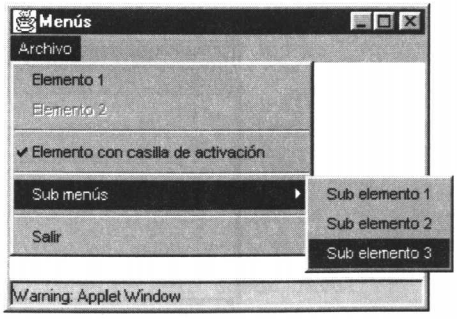


Figura 10.9. Usar submenús.

```
menuitem2 = new MenuItem("Elemento 2");
        menuitem2.addActionListener(this);
```

```
        menu.add(menuitem2);

        menu.addSeparator();

           menuitem3 = new CheckboxMenuItem("Elemento con casilla de
activación");
        menuitem3.addActionListener(this);
        menu.add(menuitem3);

        menu.addSeparator();

        submenu = new Menu("Sub menús");
        subitem1 = new MenuItem("Sub elemento 1");
        subitem2 = new MenuItem("Sub elemento 2");
        subitem3 = new MenuItem("Sub elemento 3");
        subitem1.addActionListener(this);
        subitem2.addActionListener(this);
        subitem3.addActionListener(this);
        menuitem2.addActionListener(this);
        menuitem2.addActionListener(this);
        submenu.add(subitem1);
        submenu.add(subitem2);
        submenu.add(subitem3);

        menu.add(submenu);

        menu.addSeparator();

        menuitem4 = new MenuItem("Salir");
        menuitem4.addActionListener(this);
        menu.add(menuitem4);

        menubar.add(menu);
        setMenuBar(menubar);
    }
```

A continuación se puede ver cómo se gestiona el hacer clic sobre el elemento del submenú con el método *actionPerformed:*

```
public void actionPerformed(ActionEvent event)
{
        if(event.getSource() == menuitem1){
            label.setText("Eligió el elemento 1");
        } else if(event.getSource() == menuitem2){
           menuItem2.setEnabled(false);
            label.setText("Eligió el elemento 2");
        } else if(event.getSource() == subitem1){
            label.setText("Eligió el sub elemento 1");
        } else if(event.getSource() == subitem2){
            label.setText("Eligió el sub elemento 2");
        } else if(event.getSource() == subitem3){
            label.setText("Eligió el sub elemento 3");
        } else if(event.getSource() == menuitem4){
            setVisible(false);
        }
```

```
    }
}
```

El submenú se muestra en la figura 10.9. Cuando el usuario selecciona uno de los elementos del submenú, la *applet* visualiza el elemento que fue seleccionado en la etiqueta de la ventana *frame*.

Menús emergentes

"Hey", dice el programador novato, "he visto que se puede hacer clic con el botón derecho sobre algunas aplicaciones y se visualiza un menú emergente. ¿Se puede hacer eso en Java?". "Claro", le contesta, "basta con usar la clase *PopupMenu*".

Se utiliza la clase *PopupMenu* para crear menús emergentes que no necesitan estar unidos a una barra de menú. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____java.awt.MenuComponent
     |____java.awt.MenuItem
          |____java.awt.Menu
               |____java.awt.PopupMenu
```

Tabla 10.15. Constructores de la clase *PopupMenu*.

Constructor	Descripción
PopupMenu()	Crea un nuevo menú emergente.
PopupMenu(String etiqueta)	Crea un nuevo menú emergente con el nombre dado.

Tabla 10.16. Métodos de la clase *PopupMenu*.

Método	Descripción
void addNotify()	Crea el compañero del menú emergente.
void show(Component origen, *int s, int y)*	Muestra el menú emergente en la posición x, y.

Los constructores de la clase *PopupMenu* se encuentran en la tabla 10.15 y sus métodos en la tabla 10.16.

Veamos un ejemplo. Aquí, creamos un nuevo menú emergente y le añadimos cuatro elementos (observe que además instalamos la interfaz *MouseListener* para gestionar los clics con el botón derecho):

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

/*
<APPLET
    CODE=menuemergente.class
    WIDTH=200
    HEIGHT=200 >
</APPLET>
*/

public class menuemergente extends Applet implements ActionListener,
MouseListener {

        Label label;
        PopupMenu popup;
        MenuItem menuitem1, menuitem2, menuitem3, menuitem4;

        public void init()
        {
            popup = new PopupMenu("Menu");
            menuitem1 = new MenuItem("Elemento 1");
            menuitem1.addActionListener(this);
            menuitem2 = new MenuItem("Elemento 2");
            menuitem2.addActionListener(this);
            menuitem3 = new MenuItem("Elemento 3");
            menuitem3.addActionListener(this);
            menuitem4 = new MenuItem("Elemento 4");
            menuitem4.addActionListener(this);
            popup.add(menuitem1);
            popup.addSeparator();
            popup.add(menuitem2);
            popup.addSeparator();
            popup.add(menuitem3);
            popup.addSeparator();
            popup.add(menuitem4);
            add(popup);
            label = new Label("¡Hola desde Java!");
            add(label);
            addMouseListener(this);
    }
```

Cuando el usuario hace clic con el botón derecho sobre la ventana de la *applet*, podemos utilizar el método *show* de la clase *PopupMenu* para visualizar el menú, pasándole la palabra clave *this* que apunta a la *applet* y la posición en la que se visualizará el menú emergente (aquí, visualizamos el menú emergente en la posición en la que se hace clic con el ratón):

```
public void mousePressed(MouseEvent e)
{
            if(e.getModifiers() != 0){
                popup.show(this, e.getX(), e.getY());
```

```
            }
}
```

Cuando el usuario hace clic sobre un elemento de un submenú, podemos visualizar dicho elemento con el método *actionPerformed:*

```
public void actionPerformed(ActionEvent event)
{
            if(event.getSource() == menuitem1)
                label.setText("Eligió el elemento 1");
            else if(event.getSource() == menuitem2)
                label.setText("Eligió el elemento 2");
            else if(event.getSource() == menuitem3)
                label.setText("Eligió el elemento 3");
            else if(event.getSource() == menuitem4)
                label.setText("Eligió el elemento 4");
}
```

Cuadros de diálogo

"Uh-oh", dice el programador novato, "necesito que el usuario introduzca datos, pero el gran jefe dijo que si pongo más cuadros de texto en mi programa, me arrepentiría". "¿Cuántos cuadros de texto tiene en su programa?", le pregunta. "Unos cuatrocientos", dice el programador novato. "Hmm", le dice, "bien, siempre se puede utilizar un cuadro de diálogo para obtener la entrada del usuario si no se quiere usar otro cuadro de texto". PN dice, "¡estupendo!".

El resultado se muestra en la figura 10.10. Como se puede ver en ella, el usuario puede abrir el menú emergente simplemente pulsando el botón derecho sobre la *applet.* Cuando el usuario selecciona un elemento del menú, la *applet* devuelve el elemento seleccionado. Este ejemplo está en el CD como menuemergente.java.

AWT soporta una clase especial de ventana, la clase *Dialog,* que se puede utilizar para crear cuadros de diálogo. Las ventanas que se crean con esta clase se parecen más a los cuadros de diálogo estándar que los que se crean con las ventanas *frame;* por ejemplo, las ventanas de diálogo no tienen un botón Minimizar ni Maximizar. Este es el diagrama de herencia para la clase *Dialog:*

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Container
          |____java.awt.Window
                          |____java.awt.Dialog
```

Los constructores de la clase *Dialog* se encuentran en la tabla 10.17 y sus métodos en la tabla 10.18.

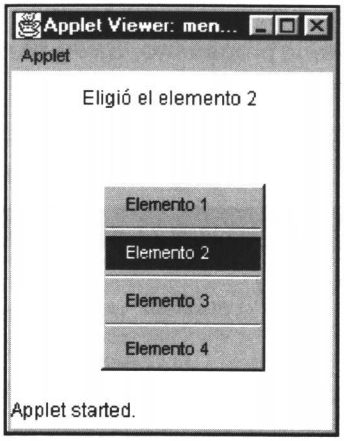


Figura 10.10. Usar menús emergentes.

Tabla 10.17. Constructores de la clase *Dialog*.

Constructor	Descripción
*Dialog(Dialog*propietario)	Crea un objeto *Dialog* inicialmente invisible y no modal sin título y el cuadro de diálogo propietario.
Dialog(Dialog propietario, *String* titulo)	Crea un objeto *Dialog,* inicialmente invisible y no modal con el propietario y título dados.
Dialog(Dialog propietario, *dialog* titulo, *boolean modal)*	Crea un objeto *Dialog,* inicialmente no visible con el propietario, título y modalidad dados.
*Dialog(Frame*propietario)	Crea un objeto *Dialog* inicialmente invisible, no modal sin título y el *frame* propietario dado.
Dialog(Frame propietario, *boolean modal)*	Crea un objeto *Dialog* inicialmente invisible, sin título y con el *frame* propietario y modalidad dados.
Dialog(Frame propietario, *String* titulo)	Crea un objeto *Dialog* inicialmente invisible, no modal con

Constructor	Descripción
	el título y el *frame* propietario dados.
Dialog(Frame propietario, *String* titulo, *boolean modal)*	Crea un objeto *Dialog,* inicialmente no visible con el propietario, título y modalidad dados.

Tabla 10.18. Métodos de la clase *Dialog.*

Método	Descripción
void addNotify()	Hace que este cuadro de diálogo sea visualizable conectándolo a un compañero nativo.
void dispose()	Elimina el cuadro de diálogo.
String getTittle()	Obtiene el título del cuadro de diálogo.
void hide()	Oculta el cuadro de diálogo.
boolean isModal()	Indica si el cuadro de diálogo es modal.
boolean isResizable()	Indica si el cuadro de diálogo puede ser redimensionado por el usuario.
protected String paramString()	Obtiene la cadena que representa el estado de este cuadro de diálogo.
void setModal(boolean b)	Especifica si este cuadro de diálogo debería ser modal.
void setResizable(boolean resizable)	Indica si este cuadro de diálogo puede ser redimensionado por el usuario.
void setTittle(String titulo)	Fija el título del cuadro de diálogo.
void show()	Hace que el cuadro de diálogo sea visible.

En estas tablas hay varias cosas que observar. Una es que se pueden crear cuadros de diálogo modales (es decir, que el usuario debe quitar el cuadro de diálogo de la pantalla antes de interactuar con el resto del programa) o cuadros de diálogo no modales, dependiendo del constructor *Dialog* que utilice o si se usa el método *setModal.* Otra puntualización es que se puede usar el método *setResizeable* para crear un cuadro de diálogo que el usuario pueda redimensionar.

Veamos un ejemplo que pone a funcionar a la clase *Dialog.* Empezaremos creando una nueva clase cuadro de diálogo llamada *okcanceldialog* que

visualiza un cuadro de texto y dos botones: Aceptar y Cancelar. Si el usuario escribe en el cuadro de texto y hace clic sobre el botón **Aceptar**, este texto aparecerá en la ventana principal de la aplicación; de lo contrario, si el usuario hace clic sobre el botón **Cancelar**, no aparecerá texto. Empezaremos por extender *okcanceldialog* de la clase *Dialog*, creando un nuevo cuadro de diálogo, y luego instalando los controles que se necesiten:

```
class okcanceldialog extends Dialog implements ActionListener
{
  Button ok, cancel;
  TextField text;

  okcanceldialog(Frame hostFrame, String title, boolean dModal)
  {
      super(hostFrame, title, dModal);
      setSize(300, 100);
      setLayout(new FlowLayout());
      ok = new Button("Aceptar");
      add(ok);
      ok.addActionListener((ActionListener) this);
      cancel = new Button("Cancelar");
      add(cancel);
      cancel.addActionListener(this);
      text = new TextField(30);
      add(text);
      data = new String("");
  }
}
```

Cuando el usuario hace clic sobre el botón **Aceptar**, el texto que el usuario escribió se almacena en un miembro de datos público llamado *data* para que sea accesible para el resto del programa en el método *actionPerformed*. Si el usuario hace clic sobre **Cancelar**, en ese miembro de datos situamos una cadena vacía. Cuando el usuario hace clic sobre cualquier otro botón, escondemos el cuadro de diálogo. Este es el código:

```
class okcanceldialog extends Dialog implements ActionListener
{
  Button ok, cancel;
  TextField text;
  public String data;

  public void actionPerformed(ActionEvent event)
  {
      if(event.getSource() == ok) {
            data = text.getText();
      } else {
            data = "";
      }
      setVisible(false);
  }
  .
```

```
    .
    .
}
```

Se puede utilizar la nueva clase *okcanceldialog* en una aplicación, visualizando un cuadro de diálogo de esa clase cuando el usuario selecciona la opción cuadro de diálogo del menú Archivo:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class dialog
{
  public static void main(String [] args)
  {
      dialogframe f = new dialogframe("Diálogos");

      f.setSize(200,200);

f.addWindowListener(new WindowAdapter() {public void
windowClosing(WindowEvent e) {System.exit(0);}});

      f.show();
  }
}

class dialogframe extends Frame implements ActionListener
{
    Menu Menu1;
    MenuBar Menubar1;
    Label label;
    MenuItem menuitem1;
    okcanceldialog dialog;

    dialogframe(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        Menubar1 = new MenuBar();

        Menu1 = new Menu("Archivo");

        menuitem1 = new MenuItem("Cuadro de diálogo...");
        Menu1.add(menuitem1);
        menuitem1.addActionListener(this);

        Menubar1.add(Menu1);

        setMenuBar(Menubar1);
        dialog = new okcanceldialog(this, "Diálogo", true);
    }
```

```
    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == menuitem1){
            dialog.setVisible(true);
            label.setText(dialog.data);
        }
    }
}
```

Observe que además se recuperan los datos del texto del cuadro de diálogo usando el miembro de datos público *data* y lo situamos en una etiqueta en la aplicación. Cuando esta aplicación se ejecuta y se hace clic sobre la opción Cuadro de diálogo... del menú Archivo, el cuadro de diálogo aparece, como se muestra en la figura 10.11.

Cuando el usuario escribe texto en el cuadro de texto del cuadro de diálogo y hace clic sobre **Aceptar**, el cuadro de diálogo se cierra y el texto aparece en la etiqueta de la ventana principal de la aplicación, como se muestra en la figura 10.12.

Hay otro tipo de cuadro de diálogo, el archivo cuadro de diálogo, que veremos a continuación.



Figura 10.11. Visualizar un cuadro de diálogo.



Figura 10.12. Visualizar texto desde el cuadro de diálogo.

Cuadros de diálogo de archivos

Se puede usar la clase especial AWT *FileDalog* para crear y visualizar un cuadro de diálogo de archivos que permita al usuario visualizar directorios y seleccionar ficheros. Este es el diagrama de herencia para esta clase:

```
java.lang.Object
|____java.awt.Component
     |____java.awt.Container
          |____java.awt.Window
               |____java.awt.Dialog
                    |____java.awt.FileDialog
```

Los campos de esta clase se encuentran en la tabla 10.19, sus constructores en la 10.20 y sus métodos en la 10.21.



Truco: observe que se pueden crear cuadros de diálogo de archivos que están especialmente indicados para abrir y guardar archivos, dependiendo de la constante de la tabla 10.19 que se utilice con el constructor *FileDialog*. Si no se especifica para qué es el cuadro de diálogo de archivos, será una versión genérica.

Tabla 10.19. Campos de la clase *FileDialog*.

Campo	Descripción
static int LOAD	Indica que el propósito del cuadro de diálogo de archivos es localizar un archivo del que se pueda leer.
static int SAVE	Indica que el propósito del cuadro de diálogo de archivos es localizar un archivo en el que se pueda escribir.

Tabla 10.20. Constructores de la clase *FileDialog*.

Constructor	Descripción
*FileDialog(Frame*padre)	Crea un cuadro de diálogo para abrir un archivo.
*FileDialog(Frame*padre, *String* titulo)	Crea un cuadro de diálogo con el título indicado para abrir un archivo.
*FileDialog(Frame*padre, *String* titulo, *int* modo)	Crea un cuadro de diálogo con el título indicado para abrir o guardar un archivo.

Tabla 10.21. Métodos de la clase *FileDialog*.

Método	Descripción
void addNotify()	Crea el compañero del cuadro de diálogo.
String getDirectory()	Obtiene el directorio de este cuadro de diálogo.
String getFile()	Obtiene el archivo seleccionado de este cuadro de diálogo.
FilenameFilter getFilenameFilter()	Determina el nombre de este cuadro de diálogo.
int getMode()	Indica si este cuadro de diálogo es para abrir un archivo o para guardarlo.
protected String paramString()	Obtiene la cadena que representa el estado de este cuadro de diálogo.
void setDirectory(String dir)	Establece que el directorio de este cuadro de diálogo es el directorio dado.
void setFile(String file)	Establece que el archivo de este cuadro de diálogo es el archivo dado.
void setFilenameFilter(Filename Filter filter)	Establece que el nombre de este cuadro de diálogo es el nombre indicado.
void setMode(int mode)	Establece el modo del cuadro de diálogo.

Este es un ejemplo en el que se permite al usuario visualizar los directorios por medio de un cuadro de diálogo. Cuando el usuario selecciona un archivo, se visualiza el nombre del mismo en la ventana principal de la aplicación.

Hacer esto es fácil; basta con crear un objeto *FileDialog*, mostrarlo con el método *setVisible*, y luego leer el nombre del archivo que el usuario seleccionó con el método *getFile:*

```
import java.awt.*
import java.awt.event.*

public class filedialog
{
  public static void main(String [] args)
  {
```

```
        dialogframe f = new dialogframe("Diálogos");

        f.setSize(200,200);

    f.addWindowListener(new   WindowAdapter()   {public   void
windowClosing(WindowEvent e) {System.exit(0);}});

        f.show();
    }
}

class dialogframe extends Frame implements ActionListener
{
    Menu Menu1;
    MenuBar Menubar1;
    Label label;
    MenuItem menuitem1;
    FileDialog dialog;

    dialogframe(String title)
    {
        super(title);
        label = new Label("¡Hola desde Java!");
        setLayout(new GridLayout(1, 1));
        add(label);
        Menubar1 = new MenuBar();

        Menu1 = new Menu("Archivo");

        menuitem1 = new MenuItem("Abrir archivo...");
        Menu1.add(menuitem1);
        menuitem1.addActionListener(this);

        Menubar1.add(Menu1);

        setMenuBar(Menubar1);
        dialog = new FileDialog(this, "Diálogo archivo", true);
    }

    public void actionPerformed(ActionEvent event)
    {
        if(event.getSource() == menuitem1){
            dialog.setVisible(true);
            label.setText("Eligió " + dialog.getFile());
        }
    }
}
```

En la figura 10.13 se muestra el resultado en Windows (por lo tanto se usa un cuadro de diálogo de archivos de Windows). En esta figura, navegamos hasta un archivo. En la figura 10.14, hemos seleccionado un archivo y cerrado el cuadro de diálogo, y la aplicación principal visualiza el archivo seleccionado.

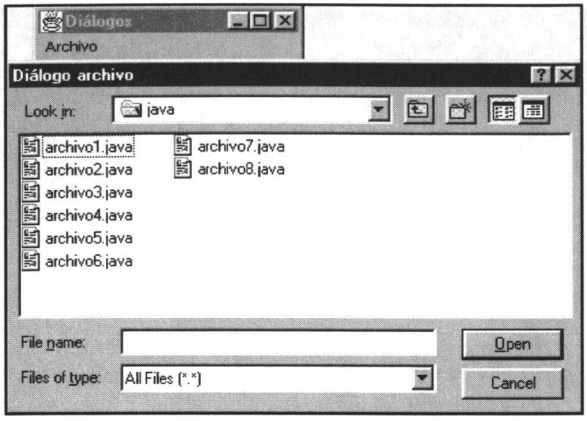


Figura 10.13. Usar un cuadro de diálogo de archivos.



Figura 10.14. Leer el nombre de un archivo desde un cuadro de diálogo de archivos.

11 *Swing:* Applets, aplicaciones y cambios de apariencia

Este capítulo inicia una sección del libro que muchos programadores estaban esperando, un debate sobre *Swing*. Demos antes un repaso a la historia de Java.

Clases *Foundation* de Java

AWT era una herramienta poderosa que impulsó la popularidad de Java. Ahora *Swing,* que tiene aproximadamente cuatro veces el número de componentes de la interfaz de usuario de AWT, es parte de la distribución estándar de Java y está tan de moda que ha llegado a desplazar a AWT. Sin embargo, AWT tuvo un avance significativo durante su existencia, aunque es, para los estándares de hoy, una implementación limitada, no diseñada para proporcionar una UI seria y básica para las necesidades de millones de programadores. El conjunto de componentes AWT no fue diseñado para soportar la popularidad con la que fue recibido y dentro de las necesidades de programación de hoy, está limitado en el rango, tiene muchos *bugs* y consume gran cantidad de recursos del sistema.

Como se mencionó anteriormente en este libro, el AWT original se tardó en escribir sólo seis semanas, fue modelado con controles HTML y asignada

una ventana del sistema operativo por componente. Dado que los programadores llegaban a utilizar gran cantidad de controles, los colaboradores empezaron a producir sus propios conjuntos de controles, lo que hizo que en *Sun Microsystems* se pusieran nerviosos. Cuando *Netscape* introdujo su librería de clases *Internet Foundation Classes* para usar con Java, éstas fueron muy populares, Sun decidió actuar y la unión de los esfuerzos de *Sun* y *Netscape* dio lugar al conjunto de componentes *Swing* como parte de las clases *Foundation* de Java (JFC).

Muchos programadores creen que JFC y *Swing* son lo mismo, pero no es así; JFC contiene *Swing* y otro gran número de elementos. Esto es lo que hay en JFC:

- *Swing:* El gran paquete UI.
- Cortar y pegar: Soporte de portapapeles.
- Accesibilidad: Ayuda a los usuarios con aquello que no es posible hacer.
- Colores del escritorio: Primero se introdujo en Java 1.1.
- Java 2D: Soporte mejorado del color, imagen y texto.
- Impresión: Originalmente disponible en Java 1.1.

Swing introdujo tres avances significativos: utiliza pocos recursos del sistema, añade gran cantidad de componentes más sofisticados y permite construir la apariencia de los programas. A continuación, veremos *Swing* con más detalle.

Swing

Swing es un conjunto de paquetes construido en la parte más alta de AWT que proporciona un gran número de clases preconstruidas (aproximadamente 250 clases y 40 componentes UI). Desde el punto de vista del programador, los componentes UI son probablemente los más interesantes, y losenumeremos aquí (observe que cada componente UI empieza con J, que es una de las razones por la que muchos programadores usan erróneamente los términos JFC y *Swing* intercambiándolos):

- *JApplet:* Versión extendida de *java.applet.Applet* que añade el soporte para los paneles base y otros paneles.
- *JButton:* Pulsador o comando de botón.

- *JCheckBox:* Casilla de activación que puede ser seleccionada o deseleccionada, de forma que visualmente se pueda saber su estado.
- *JColorChooser:* Panel de controles que permite al usuario seleccionar un color.
- *JComboBox:* Cuadro de lista desplegable, que es una combinación de cuadro de texto y lista desplegable.
- *JComponent:* Clase base para los componentes *Swing*.
- *JDesktopPane:* Contenedor usado para crear una interfaz de documentos múltiples o un escritorio.
- *JDialog:* Clase base para crear un cuadro de diálogo.
- *JEditorPane:* Componente de texto que permite al usuario editar varios tipos de contenido.
- *JFilechooser:* Permite al usuario elegir un archivo.
- *JFrame:* Una versión extendida de *java.awt.Frame* que añade soporte para los paneles base y otros paneles.
- *JInternalFrame:* Un objeto de peso ligero que proporciona muchas de las características de un *frame* peso pesado.
- *JInternalFrame.JDesktopIcon:* Representa una versión iconificada de un *JInternalFrame*.
- *JLabel:* Área visualizable para una cadena de texto corta o una imagen (o ambos).
- *JLayeredPane:* Añade capas a un contenedor *Swing*, permitiendo que los componentes se solapen unos con otros.
- *JList:* Componente que permite al usuario seleccionar uno o más objetos de una lista.
- *JMenu:* Un menú de lista desplegable que contiene *JmenuItems*, que se visualiza cuando el usuario lo selecciona en el componente *JMenuBar*.
- *JMenuBar:* Una implementación de una barra de menú.
- *JMenuItem:* Una implementación de una opción de menú.
- *JOptionPane:* Facilita la posibilidad de que emerja un cuadro de diálogo estándar.
- *JPanel:* Un contenedor genérico peso ligero.

- *JPasswordField:* Permite editar una línea simple de texto donde no muestra los caracteres originales.
- *JPopupMenu:* Un menú emergente.
- *JPopupMenu.Separator:* Separador de un menú específico emergente.
- *JProgressBar:* Componente que visualiza un valor entero en un intervalo.
- *JRadioButton:* Botón de opción que puede ser seleccionado o no, marcando su estado visualmente.
- *JRadioButtonMenuItem:* Botón de opción de un elemento de menú.
- *JRootPane:* Componente fundamental de la jerarquía del contenedor.
- *JScrollBar:* Implementación de una barra de desplazamiento.
- *JScrollPane:* Contenedor que gestiona barras de desplazamiento verticales y horizontales y las cabeceras de filas y columnas.
- *JSeparator:* Separador de menú.
- *JSlider:* Componente que permite al usuario seleccionar un valor deslizando un botón en un intervalo.
- *JSplitPane:* Divide dos componentes.
- *JTabbedPane:* Permite al usuario cambiar entre un grupo de componentes haciendo clic sobre lengüetas.
- *JTable:* Presenta datos en un formato de tabla bidimensional.
- *JTextArea:* Área de múltiples líneas que visualiza texto plano.
- *JTextField:* Permite la edición de una línea de texto sencilla.
- *JTextPane:* Componente de texto que se puede marcar con atributos.
- *JToggleButton:* Botón de dos estados.
- *JToggleButton, ToggleButtonModel:* Modelo de botón *toggle.*
- *JToolBar:* Barra de herramientas, útil para visualizar controles usados normalmente.
- *JToolBar.Separator:* Separador específico de barra de herramientas.
- *JToolTip:* Visualiza una utilidad para un componente.
- *JTree:* Visualiza un conjunto de datos jerárquicos.

- *JTree.DynamicUtilTreeNode:* Permite separar vectores/*hashtables/ arrays/*cadenas y crear nodos apropiados para los hijos.
- *JTree.EmptySelectionModel:* Modelo de elección de árbol que no permite seleccionar nada.
- *JViewport:* Vista mediante la que se ve información.
- *JWindow:* Ventana que puede visualizarse en cualquier sitio del escritorio.

Una cosa que deberíamos notar en esta lista es que todo control y contenedor AWT, excepto *JCanvas,* tiene un sustituto en *Swing;* la razón es que la clase *JPanel* ya soporta todo lo que el componente *Canvas* soportaba , por lo que *Sun* no consideró necesario añadir un componente *JCanvas* por separado.

Componentes peso pesado contra peso ligero

¿Por qué no se añadieron a AWT todas estas mejoras y nuevos componentes? Esta pregunta hace que volvamos de nuevo a ver cuál es la diferencia básica en la filosofía de los componentes AWT y *Swing*. Cada componente AWT obtiene su propia ventana de la plataforma (y por lo tanto es como si fuera un control estándar de esa plataforma). En programas extensos, todas estas ventanas ralentizan el rendimiento y consumen gran cantidad de memoria. A tales componentes se les ha denominado pesos pesados. Los controles *Swing,* por otro lado, son sencillamente dibujados como imágenes en sus contenedores y no tienen una ventana de la plataforma propia, por lo que usan bastantes menos recursos del sistema. Por este motivo, se les denomina componentes peso ligero.

Todos los componentes *Swing* se derivan de la clase *JComponent,* y esta clase es, a su vez, derivada de la clase AWT *Container,* que tiene ventana peso pesado (llamada compañero), por lo que *JComponent* es una clase peso ligero. Como se verá en este capítulo, *JComponent* añade una cantidad tremenda de soporte de programación a la clase de componentes AWT. Observe que dado que los componentes *Swing* se derivan de *JComponent,* y que *JComponent* se deriva de la clase *Container* de AWT, todos los componentes *Swing* son componentes AWT y se pueden mezclar controles AWT con controles *Swing* en los programas.

Por lo tanto, dado que los controles *Swing* se dibujan en su contenedor, se pueden obtener resultados extraños porque los controles Java aparecerán por encima de ellos.

Ahora todos los componentes *Swing* son pesos ligeros; para visualizar todo en un entorno de ventanas, son necesarias algunas ventanas del sistema operativo, no sólo para dibujar controles de peso ligero. Por esa razón, *Swing* soporta estas clases peso ligero: *JFrame, JDialog, JWindow* y *JApplet.*

Igual que construimos *applets* AWT usando la clase *Applet* y aplicaciones AWT usando la clase *Frame,* construiremos *applets Swing* en la clase *JApplet* y aplicaciones *Swing* usando la clase *Jframe.* El hecho de que *Swing* esté construido sobre AWT no significa nada para el programador, y los contenedores peso pesado *Swing* de hecho tienen una estructura compleja para ser usados. De hecho, *Swing* con frecuencia tiene el sentimiento de que algo es añadido por una tercera parte, no por Sun. Por ejemplo, para pintar componentes, no se sobrescribe el método *paint* nunca más, porque *Swing* necesita hacer eso para dibujar los bordes y todo lo demás, y tendremos acceso directo a las partes del programa en las que se dibujan los menús de cuadros de diálogo. Todo requiere un esfuerzo adicional de programación para pasar de AWT a *Swing,* como veremos en este capítulo.

Características *Swing*

Además del gran *array* de componentes en *Swing* y el hecho de que sean peso ligero, *Swing* introduce muchas otras innovaciones. Estas son algunas de ellas:

- Bordes: Se pueden dibujar bordes alrededor de los componentes, de diferentes, utilizando el método *setBorder.*
- *Debugging* de gráficos: Se puede usar el método *setDebuggingGraphicsOptions* para iniciar el *debugging* de gráficos, lo que significa, entre otras cosas, que se puede ver cada línea cuando se está dibujando y hacerla un flash.
- *Facilidad de operaciones sin ratón:* Ahora es fácil conectar las teclas a los componentes.
- *Tooltips:* Se puede usar el método *JComponent.setToopTipText* para dar a los componentes un *tooltip,* una de esas pequeñas ventanas con texto explicativo, que aparece cuando se mueve el ratón alrededor de un componente.
- Facilidad de desplazamiento: Ahora se puede conectar el desplazamiento a varios componentes, algo que era imposible con AWT.
- Apariencia: Se puede establecer la apariencia de las *applets* y aplicaciones como cualquiera de los tres estándares: W*indows, Motif (Unix)* o *Metal* (la apariencia estándar de *Swing*).

- Nuevos gestores de esquemas: *Swing* introduce los gestores *BoxLayout* y *Overlayout.*

De todas esta innovaciones, la apariencia es probablemente la más importante, porque permite seleccionar el estilo de la apariencia del programa. En AWT, los controles se construían en ventanas de la plataforma, lo que hizo que los programas AWT tuvieran una apariencia dependiente de la plataforma, y esto no era una buena idea para un lenguaje que estaba orgulloso de ser multiplataforma. Por ejemplo, ya que fuentes y controles diferentes se usan en diferentes plataformas, el esquema de los programas podría aparecer diferente en distintas plataformas. *Swing* introduce la apariencia *Metal*, que es la nueva apariencia de Java, y será igual en todas las plataformas. Adicionalmente, se puede dar a los programas una apariencia de *Windows* o *Motif.* Veremos este proceso en este capítulo.



Truco: *Sun* ha creado varias apariencias que no se entregan con JDK, incluyendo una para *Apple Macintosh* y otra independiente de la plataforma llamada *Organic.*

Entre AWT y *Swing* hay dos diferencias de programación esenciales. Una es que cuando se crean *applets Swing* y aplicaciones se trabaja con paneles, y la otra es la arquitectura de programación Modelo-Vista-Controlador. Primero revisaremos cómo *Swing* funciona con paneles, ya que hay que entender cómo lo hacen los paneles antes de fijarlos en cualquier sitio en *Swing.*



Truco: aquí indicamos otra diferencia entre AWT y *Swing:* Cuando se vuelven a dibujar elementos en la pantalla de AWT, primero se llama al método *update* para dibujar el fondo del elemento, y los programadores con frecuencia sobrescriben *update* para llamar al método *paint* directamente, evitando así las fluctuaciones. En *Swing,* por otro lado, el método *update* no vuelve a dibujar el fondo del elemento, ya que los componentes pueden ser transparentes; en su lugar, *update* llama a *paint* directamente.

Utilizar paneles en la programación de gráficos

Las clases que son la base de las *applets* y aplicaciones, *JApplet* y *JFrame,* tienen un objeto hijo que hace todo el trabajo de gráficos, y este objeto es de la clase *JRootPane.* En la programación AWT, directamente, sólo se pueden

añadir componentes a las *applets* o ventanas *frame*, pero en *Swing* es un poco más complejo. Aunque realmente se pueden añadir componentes directamente a *JFrame* o *JApplet*, se obtendrá un error a menos que primero se desactive la verificación de errores, porque se espera que los añada al *content Pune* del objeto *JRootPane*.

Esta es la estructura de un *JRootPane:* el panel tiene otros dos, un *layered pane* y un *glass pane*. El *glas pane* es realmente un objeto *Component* de AWT que pasa por todo el resto y que intercepta los eventos de ratón cuando se hace que este panel sea visible (aunque generalmente es transparente) y añade un *Mouse listener*. El *layered pane*, que es de la clase *JLayeredPane*, es donde tiene lugar la mayoría de las acciones. Hay capas específicas que tienen acceso, que visualizan menús y cuadros de diálogo cuando se abren. Desde el punto de vista del programador, probablemente los elementos más interesantes en el *layered pane* son la barra de menús y el área de trabajo. El *content pane* es realmente un objeto de la clase AWT *Container* (no hay clase *JContentPane*), y es donde las *applets* y las aplicaciones visualizan sus componentes. En la práctica, esto significa que cuando se añaden componentes a un programa, se les añade al área de trabajo, que quiere decir que primero hay que obtener una referencia de dicho panel. Si se tiene una barra de menú en el programa, soportado con el objeto *JMenuBar*, se visualiza sobre el *content pane*.



Truco: los *content pane* de *Swing* utilizan el gestor *border layout* por defecto, mientras que las *applets* AWT utilizan *flow layout* por defecto y las ventanas *frame AWT*, el *border layout*.

Arquitectura Modelo-Vista-Controlador

Hay un último punto que tratar antes de pasar a la programación *Swing:* la arquitectura Modelo-Vista-Controlador (MVC). Esta arquitectura es el corazón de la programación de componentes UI de *Swing*, y la comprensión del término es fundamental. Esto es lo que este término significa: el modelo de un componente está donde están almacenados sus datos, como el estado de un botón o los elementos de una lista. La vista es la representación en pantalla del componente, como la forma en que aparece un botón o una lista. Finalmente, el controlador es la parte del componente que gestiona la entrada, como los clics del ratón.

Dividir grandes programas en vistas y documentos, como en Microsoft Visual C++, se ha convertido en algo estándar. En este caso, una vista pro-

porciona una ventana en un documento, que es donde los datos del programa se almacenan; múltiples vistas pueden proporcionar múltiples ventanas en el documento. *Swing,* inspirado en el lenguaje Smalltalk, va más adelante y basa sus componentes UI en la arquitectura MVC. Como se puede imaginar, en *Swing* es útil separar la vista del modelo, donde la apariencia de los componentes pueden cambiar con unas pocas líneas de código. Pronto verá más sobre la arquitectura MVC.

Ahora que hemos dado una pasada rápida por *Swing,* es hora de pasar a la siguiente sección.

Trabajar con *Swing*

El programador novato aparece y dice, "De acuerdo, estoy preparado para empezar a trabajar con *Swing*. ¿Por dónde empiezo?" "Bien", le contesta, "probablemente por la clase *JComponent*".

La clase *JComponent* es la base de todos los componentes *Swing*. Es una clase peso ligero derivada de la clase AWT *Container*. Formalmente hablando, *JComponent* es realmente la clase *javax.swing.JComponent,* porque *javax* es el paquete que contiene *Swing*. Este es el diagrama de herencia para *JComponent:*

```
java.lang.Object
|____java.awt.Component
          |____java.awt.Container
                    |____javax.swing.JComponent
```

La clase *JComponent* añade bastantes cosas a la clase AWT *Container,* como la posibilidad de dibujar bordes predefinidos alrededor de componentes UI, añadir objetos *PropertyChangeListener,* que son notificados cuando se cambia el valor de la propiedad, y mucho más.

Para referencias, se debería echar un vistazo a las tablas 11.1, 11.2 y 11.3, que contienen los campos, constructores y métodos, respectivamente, de esta gran clase.

Tabla 11.1. Campos de la clase *JComponent.*

Campo	Descripción
protected AccessibleContext accessibleContext	Soporte de acceso.
protected EventListenerList listenerList	Lista de los *listeners* de eventos actuales.

Campo	Descripción
static String TTOL_TIP_TEXT_KEY	Comentario que se visualiza cuando el cursor está sobre el componente.
protected ComponentUI ui	Interfaz de usuario.
static int UNDEFINED_CONDITION	Constante usada para indicar que no se ha definido ninguna condición.
static int WHEN_ANCESTOR_OF _ FOCUSED_COMPONENT	Usado por *registerKeyboardAction()*, indicando que se debería llamar al comando cuando el componente que recibe es un progenitor del componente que tiene el foco.
static int WHEN_FOCUSED	Constante usada por *registerKeyboardAction()*, indicando que se debería llamar al comando cuando el componente tenga el foco.
static int WHEN_IN_FOCUSED_ WINDOW	Constante usada por *registerKeyboardAction()*, indicando que se debería llamar al comando cuando el componente tenga el foco.

Tabla 11.2. Constructor de la clase *JComponent*.

Constructor	Descripción
JComponent()	Constructor por defecto de *JComponent*.

Tabla 11.3. Métodos de la clase *JComponent*.

Método	Descripción
void addAncestorListener(AncestorListener listener)	Registra el *listener* para que reciba *AncestorEvents*.
void addNotify()	Notificación de que este componente tiene ahora un componente padre.
void addPropertyChangeListener (PropertyChangeListener listener)	Añade un *PropertyChangeListener* a la lista de *listener*.

Método	Descripción
void addPorpertyChangeListener(String propertyName, PropertyChangeListener listener)	Añade un *PropertyChangeListener* para una propiedad específica.
void addVetoableChangeListener (VetoableChangeListener listener)	Añade un *VetoableChangeListener* a la lista de listener.
void computeVisibleRect(Rectangle visibleRect)	Obtiene el rectángulo visible del componente.
boolean contains(int x, int y)	Determina si el componente contiene el punto dado.
JtoolTip createToolTip()	Obtiene la instancia de *JtoolTip* que debería usarse para visualizar el *tooltip.*
void firePropertyChange(String propertyName, boolean oldValue, boolean newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, byte oldValue, byte newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, char oldValue, char newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, double oldValue, double newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, float oldValue, float newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, int oldValue, int newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, long oldValue, long newValue)	Reporta un cambio de propiedad de fronteras.
void firePropertyChange(String propertyName, Object oldValue, Object newValue)	Proporciona soporte para reportar los cambios en las propiedades de fronteras.
void firePropertyChange(String propertyName, short oldValue, short newValue)	Reporta un cambio de propiedad de fronteras.
protected void fireVetoableChange(String propertyName, Object oldValue, Object newValue)	Proporciona soporte para reportar los cambios de propiedades.

Método	Descripción
AccessibleContext getAccessibleContext()	Obtiene el *AccessibleContext* asociado con este *JComponent.*
ActionListener getActionForKeyStroke (KeyStroke aKeyStroke)	Obtiene el objeto que ejecutará la acción registrada para una combinación de teclas dada.
float getAlignmentX()	Obtiene la alineación vertical.
float getAlignmentY()	Obtiene la alineación horizontal.
boolean getAutoscrolls()	Devuelve verdadero si este componente automáticamente desplaza su contenido cuando se arrastra.
Border getBorder()	Obtiene el borde de este componente o nulo si no está el borde.
Rectangle getBounds(Rectange rv)	Almacena las fronteras de este componente en rv y devuelve rv.
Object getClientProperty(Object key)	Obtiene el valor de la propiedad con la tecla especificada.
protected Graphics getComponent-Graphics(Graphics g)	Obtiene el objeto gráfico usado para pintar este componente.
int getConditionForKeyStroke(Key-Stroke aKeyStroke)	Obtiene la condición que determina si ocurre una acción para la tecla aceleradora especificada.
int getDebugGraphicsOptions()	Obtiene el estado del *debugging* de gráficos.
Graphics getGraphics()	Obtiene el contexto de gráficos de este componente.
int getHeight()	Obtiene la altura actual de este componente.
Insets getInsets()	Si se ha establecido un borde para este componente, el método obtiene los intercalados. De lo contrario llama a *super. get-insets.*
Insets getInsets(Insets insets)	Obtiene un objeto *Insets* para este componente.

Método	Descripción
Point getLocation(Point rv)	Almacena el origen x,y de este componente en rv y devuelve rv.
Dimension getMaximumSize()	Obtiene el tamaño máximo del componente.
Dimension getMinimumSize()	Obtiene el tamaño mínimo del componente.
Component getNextFocusableComponent()	Obtiene el siguiente componente que debe tener el foco o null.
Dimension getPreferredSize()	Si el tamaño preferido se ha establecido a un valor no nulo, este método lo devuelve.
KeyStroke[] getRegisteredKeyStrokes()	Obtiene las teclas aceleradoras que inician acciones.
JRootPane getRootPane()	Obtiene el progenitor de un componente.
Dimension getSize(Dimension rv)	Almacena la anchura y altura de este componente en rv y devuelve rv.
Point getToolTipLocation(Mouse Event event)	Obtiene la posición del *tooltip* en el componente.
String getToolTipText()	Obtiene la cadena *tooltip* que se ha establecido con *setToolTipText().*
String getToolTipText(MouseEvent event)	Obtiene la cadena a utilizar por el *tooltip* en ese evento.
Container getTopLevelAncestro()	Obtiene el primer antecesor del componente.
String getUIClassID()	Obtiene la clave *UIDefaults* usada para encontrar el nombre de la clase *swing.plaf.ComponentUI.*
Rectangle getVisibleRect()	Obtiene el rectángulo visible del componente.
int getWidth()	Obtiene la anchura actual.
int getX()	Obtiene la coordenada x actual del origen de este componente.

Método	Descripción
int getY()	Obtiene la coordenada y actual del origen de este componente.
void grabFocus()	Fija el foco en el componente.
boolean hasFocus()	Devuelve verdadero si este componente tiene el foco del teclado.
boolean isDoubleBuffered()	Indica si el componente receptor debería usar un *buffer* para pintar.
boolean isFocusCycleRoot()	Devuelve verdadero si el componente es la raíz de un árbol de componente con un ciclo de foco.
boolean isFocusTraversable()	Especifica si este componente puede recibir el foco.
static boolean isLightweightComponent (Component c)	Devuelve verdadero si este componente es peso ligero.
boolean isManagingFocus()	Devuelve verdadero si el componente gestiona el foco.
boolean isOpaque()	Devuelve verdadero si el componente es opaco.
boolean isOptimizedDrawingEnabled()	Devuelve verdadero si el componente titula a sus hijos.
boolean isPaintingTile()	Devuelve verdadero si el componente está pintando un entramado.
boolean isRequestFocusEnabled()	Devuelve verdadero si el componente receptor puede obtener el foco.

Preparar para crear un *applet Swing*

"De acuerdo", dice el programador novato, "estoy preparado para empezar a crear *applets* usando *Swing*. ¿Cómo lo hago?" "Con la clase *JApplet*", le responde, "coja una silla y veámoslo".

La clase *JApplet* es un contenedor peso pesado que es la base de las *applets Swing* y una extensión de la clase *Applet* AWT. Esta clase tiene un

hijo, un objeto de la clase *JRootPane,* y es donde se continúa el dibujo y donde se añaden los componentes. Este es el diagrama de herencia de *JApplet:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____java.awt.Panel
                        |____java.applet.Applet
                                |____javax.swing.JApplet
```

Los campos de esta clase se encuentran en la tabla 11.4, su constructor en la 11.5 y sus métodos en la 11.6.

Para arrancar un *applet Swing,* basta derivar una clase de la clase *JApplet* como sigue (observe que importamos *javax.swing.** para usar *JApplet*):

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

/*
<APPLET
    CODE=applet.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class applet extends JApplet
{
   .
   .
   .
}
```

Desafortunadamente, en este punto, se complican más las cosas (cuando se quiere dibujar en un *applet* o añadir controles en él), porque normalmente se trabaja con el *content pane* en la *applet* en lugar de hacerlo en la *applet* en sí misma.

Tabla 11.4. Campos de la clase *JApplet.*

Campo	Descripción
protected AccessibleContext accessibleContext	Accesibilidad a un contexto.
protected JRootPane rootPane	Panel raíz.
protected boolean rootPanecheckingEnabled	Indica si se producirán errores al intentar añadir componentes al *root pane.*

Tabla 11.5. Constructor de la clase *JApplet*.

Constructor	Descripción
JApplet()	Crea una instancia a un *applet Swing.*
protected void addImpl(Component comp, Object constraints, int indice)	Añade hijo al *content pane.*
protected JRootPane createRootPane()	Lo llaman los métodos del constructor para crear el *root pane* por defecto.
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible asociado con esta *JApplet.*
Container getContentPane()	Obtiene el objeto *contentPane* para esta *applet.*
Component getGlassPane()	Obtiene el objeto *glassPane* para esta *applet.*
JMenuBar getJMenuBar()	Obtiene la barra de menú para esta *applet.*
JLayeredPane getLayeredPane()	Obtiene el objeto *layeredPane* para esta *applet.*
JRootPane getRootPane()	Obtiene el objeto *rootPane* para esta *applet.*
protected boolean isRootPaneChecking-Enabled()	Devuelve verdadero si la verificación del *root pane* está habilitado.
protected String paramString()	Obtiene una representación en cadena de esta *JApplet.*
protected void processKeyEvent (KeyEvent e)	Procesa eventos clave que ocurren en este componente enviándolos a los objetos *KeyListener.*
void remove(Component comp)	Elimina el componente especificado de este contenedor.
void setContentPane(Container contentPane)	Fija la propiedad *contentPane.*
void setGlassPane(Component glass-Pane)	Obtiene la propiedad *glassPane.*
void setJMenuBar(JMenuBar menuBar)	Obtiene la barra de menú de esta *applet.*

Constructor	Descripción
void setLeyeredPane(JLayeredPane layeredPane)	Obtiene la propiedad *layeredPane.*
void setLayout(LayoutManager manager)	Fija el esquema de este *content pane.*
protected void setRootPane(JRootPane root)	Fija la propiedad *rootPane.*
protected void setRootPaneChecking-Enabled(boolean enabled)	Si es verdadero, llama a *add* y *setlayout* causará una excepción.
void update(Graphics g)	Llama *paint(g).* No vuelve a dibujar el fondo.

Tanto *JApplet* como *JFrame* tienen un objeto hijo, un objeto de la clase *JRootPane,* y el *content pane* es parte de *root pane.* Echaremos un vistazo a los *root panes* en el siguiente punto.

Comprender los *root panes*

La clase *JRootPane* es la clase que gestiona la apariencia de los objetos de *JApplet* y *JFrame.* En particular, el *root pane* contiene *glass pane, content pane* y *layered pane,* que veremos en la programación *Swing.*

Este es el diagrama de herencia para *JRootPane:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.JRootPane
```

Los campos de la clase *JRootPane* los encontrará en la tabla 11.7, su constructor en la 11.8 y sus métodos en la 11.9.

Tabla 11.7. Campos de la clase *JRootPane.*

Campo	Descripción
protected Container contentPane	El *content pane.*
protected JButton defaultButton	Botón que está activado cuando el panel tiene el foco y se produce una acción, como es presionar la tecla Intro.

Campo	Descripción
protected javax.swing.JRootPane. DefaultAction defaultPressAction	Acción que se produce cuando se presiona el botón por defecto.
protected javax.swing.JRootPane. DefaultAction defaultReleaseAction	Acción que se produce cuando el botón, por defecto, se suelta.
protected Component glassPane	*Glass pane* que cubre la barra de menú y el *content pane,* por lo que pueden interceptar los movimientos de ratón.
protected JLayeredPane layeredPane	El *layered pane* que gestiona la barra de menú y el *content pane.*
protected JMenuBar menuBar	Barra de menú.

Tabla 11.8. Constructor de la clase *JRootPane.*

Constructor	Descripción
JRootPanel()	Crea un componente *JRootPane,* estableciendo sus componentes *glassPane, LayeredPane* y *contentPane.*

Tabla 11.9. Métodos de la clase *JRootPane.*

Método	Descripción
protected void addImpl(Component comp, Object constraints, int indice)	Sobrescrito por *Swing.*
void addNotify()	Registra un nuevo *root pane.*
protected Container createContentPane()	Llamado por los métodos constructores para crear el *content pane* por defecto.
protected Component createGlassPane()	Llamado por los métodos constructores para crear el *glass pane* por defecto.
protected JLayeredPane createpane. LayeredPane()	Llamado por los métodos constructores para crear el *layered* por defecto.
protected LayoutManager createRootLayout()	Llamado por los métodos constructores para crear el *layout manager* por defecto.

Método	Descripción
Component findComponentAt(int x, int y)	Localiza el componente hijo visible que contiene la posición especificada.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible.
Container getContentPane()	Obtiene el *content pane.*
JButton getDefaultButton()	Obtiene el botón por defecto.
Component getGlassPane()	Obtiene el *glass pane.*
JMenuBar getJMenuBar()	Obtiene la barra de menú del *layered pane.*
JLayeredPane getLayeredPane()	Obtiene el *layered pane* usado por el *root pane.*
JMenuBar getMenuBar()	Obsoleto. Reemplazado por *getJMenubar().*
boolean isFocusCycleRoot()	Forma la raíz de un ciclo de foco.
boolean isValidateRoot()	Si un descendiente de esta *JRootPane* llama a *revalidate,* la validación tendrá lugar de aquí hacia abajo.
protected String paramString()	Obtiene una representación en cadena de *root pane.*
void removeNotify()	Elimina el registro desde *SystemEventQueueUtils.*
void setContentPane(Container content)	Fija el *content pane.*
void setDefaultButton(Jbutton defaultButton)	Fija el botón actual por defecto.
void setGlassPane(Component glass)	Fija un componente especificado para ser el *glass pane* para este *root pane.*
void setJMenuBar(JMenuBar menu)	Añade o cambia la barra de menú usada en el *layered pane.*
void setLayeredPane(JLayeredPane layered)	Fija el *layered pane* para el *root pane.*
void setMenuBar(JMenuBar menu)	Obsoleto. Reemplazado por *setJMenuBar.*

Para construir *applets* y aplicaciones, generalmente se trabaja con el *content pane* del *root pane*. El *content pane* en sí mismo, sin embargo, es parte de otro panel, el *layered pane*. Lo veremos en el siguiente punto.

Comprender *layered panes*

Los *layered pane* del *root pane* gestionan los componentes actuales que aparecen en las *applets* y aplicaciones, incluyendo barras de menú y el *content pane*. Este es el diagrama de herencia para la clase de *layered pane, JLayeredPane:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.JLayeredPane
```

Por conveniencia, *JLayeredPane* divide el rango en varias capas diferentes. Poner un componente en una de estas capas hace que sea más fácil asegurarse de que los componentes se solapen adecuadamente, sin tener que preocuparse de especificar los miembros para cada espesor:

- *DEFAULT_LAYER:* La capa estándar y la de más al fondo, donde van los componentes.
- *PALETTE_LAYER:* La capa de paleta se pone como defecto y es útil para barras de herramientas y paletas flotantes.
- *MODAL_LAYER:* Capa usada para cuadros de diálogo modales.
- *POPUP_LAYER:* Capa de menú emergente que se visualiza sobre la capa de diálogo.
- *DRAG_LAYER:* Cuando se arrastra un componente, asignándolo a la capa de arrastre, se asegura de estar situado sobre otro componente en el contenedor.

Para la reposición de un componente dentro de su capa, se puede usar los métodos *JLayeredPane, moveToFront, moveToBack* y *setPosition.* También se puede utilizar el método *setLayer* usado para cambiar la capa actual del componente. Los campos de la clase *JLayeredPane* se encuentran en la tabla 11.10, su constructor en la 11.11 y sus métodos en la 11.12.

Como es típico que las operaciones de dibujo vayan en *applets* y aplicaciones, la parte más importante del *layered pane* es el *content pane.* Lo veremos en el siguiente punto.

Tabla 11.10. Campos de la clase *JLayeredPane.*

Campo	Descripción
static Integer DEFAULT_LAYER	Capa por defecto.
static Integer DRAG_LAYER	Capa de arrastre.
statis Integer FRAME_CONTENT_LAYER	Capa *frame content.*
static String LAYER_PROPERTY	Propiedad de frontera.
static Integer MODAL_LAYER	Capa modal.
static Integer PALETTE_LAYER	Capa de paleta.
static Integer POPUP_LAYER	Capa de menú emergente.

Tabla 11.11. Constructor de la clase *JLayeredPane.*

Constructor	Descripción
JLayeredPane()	Crea un nuevo objeto *JLayeredPane.*

Tabla 11.12. Métodos de la clase *JLayeredPane.*

Método	Descripción
protected void addImpl(Component indexcomp, Object constraints, int indice)	Añade el componente indicado a este contenedor.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible.
int getComponentCountInLayer(int layer)	Obtiene el número de hijos en la capa indicada.
Component[] getComponentInLayer (int layer)	Obtiene un *array* de los componentes en la capa indicada.
protected Hashtable getComponentToLayer()	Obtiene el *hashtable* que hace corresponder los componentes con las capas.
int getIndexOf(Component c)	Obtiene el atributo capa para el componente indicado.
static int getLayer(JComponent c)	Obtiene la propiedad de capa para un *JComponent* y no causa ningún efecto lateral como *setLayer().*

Método	Descripción
static JLayeredPane getLayeredPane-Above(Component c)	Obtiene el primer *JLayered-Pane,* que contiene el componente indicado.
protected Integer getObjectForLayer(int layer)	Obtiene el objeto *Integer* con una capa indicada.
int getPosition(Component c)	Obtiene la posición relativa del componente dentro de su capa.
int highestLayer()	Obtiene el valor de la capa más alta desde el hijo actual.
protected int insertIndexForLayer(int layer, int posición)	Determina la propia ubicación en la que se inserta un nuevo hijo basado en la capa y la posición.
boolean isOptimizedDrawingEnabled()	Devuelve falso si los componentes en el panel pueden solaparse.
int lowestLayer()	Obtiene el valor de la capa más baja desde el hijo actual.
void moveToBack(Component c)	Mueve el componente a la parte superior de los componentes en su capa actual.
void paint(Graphics g)	Pinta el *layered pane* usando un contexto gráfico.
protected String paramString()	Obtiene una representación en cadena de este*JLayeredPane.*
static void putLayer(JComponent c, int layer)	Fija la propiedad de capa en un *JComponent.*
void remove(int indice)	Retira el componente indexado de este panel.
void setLayer(Component c, int layer, int posición)	Fija el atributo de la capa para el componente indicado y además fija su posición dentro de esta capa.
void setPosition(Component c, int posición)	Mueve el componente a la posición dentro de su capa actual.

Comprender los *content panes*

Generalmente, los controles y los gráficos se sitúan en un *content pane* de un *applet* o aplicación. Se puede esperar que el *content pane* lo soporte su propia clase, pero de hecho, un *content pane* es sólo un objeto *Container* (que es una clase peso ligero). Se puede tener acceso al *content pane* en el *layered pane* de un *applet* o aplicación, usando el método *getContentPane* de las clases *JApplet* y *JFrame*.

La importancia del *content pane* se debe a que es el lugar donde generalmente se añaden componentes a las *applets* y aplicaciones, así como donde se dibujan gráficos; esto marca una gran diferencia con la programación AWT. Otro punto importante es conocer qué *content panes* usa, por defecto, *border layout*. Para ver cómo funciona esto realmente en el código, echemos un vistazo al siguiente punto.

Crear un *applet Swing*

"De acuerdo", dice el programador novato, "estoy preparado para crear un *applet Swing* y añadir elementos al *content pane*. ¿Cómo se hace esto?" "Siéntese y discutámoslo", le contesta.

Pintar en *Swing* frente a AWT

Cuando hemos visto el contenido de AWT antes en este capítulo, quizás estuviera esperando ver un método *paint* para gestionar las operaciones de pintura y los gráficos en *applets* y aplicaciones *Swing*. De hecho, hay un método *paint*, pero no debería sobrescribirlo a menos que realmente sepa lo que está haciendo, porque *Swing*, en sí mismo, utiliza ese método para dibujar los bordes alrededor del componente, así como para dibujar cualquier componente hijo en el componente, entre otras tareas.

En lugar de usar el método *paint* para las operaciones de pintura, ahora se utiliza el método *paintComponent*. Éste tiene algunas ramificaciones al crear *applets* y aplicaciones, comparado con la programación AWT. En lugar de sobrescribir, sólo, el método *paint* de la *applet* o ventana *frame*, ahora hay que crear un componente para pintar en él. Esto se debe hacer porque el *content pane* en un programa no es una clase en la que se puedan sobrescribir métodos, sino que es un objeto (una alternativa es crear una clase *content pane* personalizada y usar el método *setContentPane* para instalarla en el

programa). Además, observe que cuando se trabaja con *paintComponent,* lo primero que se debería hacer es llamar a *super.paintComponent,* el método *paintComponent* de la super clase.

Visualizar controles en *Swing* frente a AWT

Para añadir controles a un *content pane,* se usa el método *add* del *content pane.* Tanto en *applets* como en aplicaciones se usa por defecto el *border layout,* a diferencia de la programación AWT, en la que las *applets* usan *flow layout* y las ventanas *frame, border layout.*

Para añadir controles a un *content pane,* primero hay que fijar el *layout manager* como se quiera, o usar el *border layout* por defecto. Para añadir controles, usaremos el método *add,* como se hace en la programación AWT.

Veamos un ejemplo. En este caso, visualizamos el texto "¡Hola desde Swing!" en un *applet.* ¿Cómo se visualiza el texto? No podemos visualizarlo en el *content pane* por defecto de la *applet,* ya que es un objeto, no una clase, por lo que no podemos sobrescribir su método *paintComponent.* Por otro lado, no se puede añadir un panel, representado por la clase *JPanel,* a la *applet* para que cubra el *content pane.*

Además se puede dibujar en ese panel sobrescribiendo su método *paintComponent.* Observe que se podría crear una nueva clase *content pane,* sobrescribir su método *paintComponent,* e instalarlo con el método *setContentPane,* pero *JPanel* generalmente se utiliza para crear *content panes* de cualquier forma.



Truco: si quiere saber exactamente qué valores se usaron para construir un componente visual tal como *JApplet,* se le da el foco, se presionan las teclas **Mayús-Control-F1** y se va a la consola para tener una visión completa.

Usar la clase *JPanel*

La clase *JPanel* es la clase contenedora para cualquier propósito de *Swing,* y es importante conocerla. Este es el diagrama de herencia de *JPanel:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.JPanel
```

Al contrario que los contenedores peso pesado de *Swing*, *JPanel* usa por defecto un *flow layout*. Los constructores de la clase *JPanel* se encuentran en la tabla 11.13 y sus métodos en la tabla 11.14.

Tabla 11.13. Constructores de la clase *JPanel*.

Constructor	Descripción
JPanel()	Construye un nuevo *JPanel* con un doble *buffer* y un *flow layout*.
JPanel(boolean isDoubleBuffered)	Construye un nuevo *JPanel* con un *flow layout* y la estrategia de *buffer* especificada.
JPanel(LayoutManager layout)	Construye un nuevo *JPanel* con el *layout manager* especificado.
JPanel(LayoutManager layout, boolean isDoubleBuffered)	Construye un nuevo *JPanel* con el *layout manager* y la estrategia de *buffer* especificados.

Tabla 11.14. Métodos de la clase *JPanel*.

Métodos	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
String getUIClassID()	Obtiene una cadena que especifica el nombre de la clase que devuelve este componente.
protected String paramString()	Obtiene una representación en cadena de *JPanel*.
void updateUI()	Se le llama cuando la apariencia ha cambiado.

A continuación crearemos una clase *JPanel* que visualiza el texto "¡Hola desde Swing!" (observe que a este panel se le está dando un fondo blanco y que se sobrescribe el método *paintComponent;* se le pasa un objeto AWT *Graphics*, que se puede usar para escribir el texto):

```
class jpanel extends JPanel
{
    jpanel()
    {
        setBackground(Color.white);
```

```
    }

    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("¡Hola desde Swing!", 0, 60);
    }
}
```

Ahora se puede añadir un objeto de esta nueva clase al *content pane* de un *applet* usando el método *getContentPane* de la clase *JApplet*, y después usar el método *add* del *content pane* para añadir el panel:

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

/*
<APPLET
    CODE=applet.class
    WIDTH=300
    HEIGHT=200 >
<APPLET
*/

public class applet extends JApplet
{
    jpanel j;

    public void init()
    {
        Container contentPane = getContentPane();

        j = new jpanel();
        contentPane.add(j);
    }
}

class jpanel extends JPanel
{
    jpanel()
    {
        setBackground(Color.white);
    }

    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("¡Hola desde Swing!", 0, 60);
    }
}
```

Ahora se puede compilar y ejecutar esta *applet*, como se muestra en la figura 11.1. Este ejemplo está en el CD que acompaña a este libro como *applet.java*.

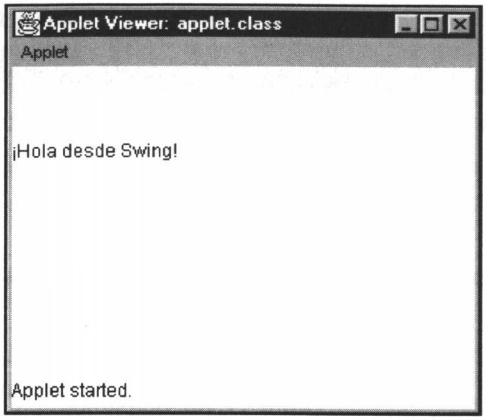


Figura 11.1. Un *applet Swing.*

Crear una aplicación *Swing*

"De acuerdo", dice el programador novato, "estoy preparado para crear una aplicación usando *Swing*. ¿Por dónde empiezo?" "Creando una ventana *frame*", le dice. "¿Usando la clase *Frame*?," pregunta NP. "No", le contesta, "la clase *JFrame*".

Al igual que en las aplicaciones AWT generalmente usamos una ventana *frame,* cuando se crean aplicaciones en *Swing* se usa un objeto *JFrame,* que tiene un *root pane* hijo.

Este es el diagrama de herencia para *JFrame* (observe que se deriva de la clase *Frame* y es un contenedor peso pesado):

```
java.lang.Object
|____java.awt.Component
          |____java.awt.Container
                    |____java.awt.Frame
                              |____javax.swing.JFrame
```

Los campos de la clase *JFrame* se encuentran en la tabla 11.15, sus constructores en la 11.16 y sus métodos en la tabla 11.17. Observe que el *layout* por defecto en un *JFrame* es un *border layout.*



Truco: si se quiere averiguar exactamente qué valores se utilizaron para construir un componente visual como un *JFrame,* se le da el foco, presionando las teclas **Mayús-Control-F1**, y se va a la consola para tener una visión completa.

Tabla 11.15. Campos de la clase *JFrame*.

Campo	Descripción
protected AccessibleContext accessibleContext	Contexto accesible.
protected JRootPane rootPane	*Root pane* que gestiona el *content pane*, barra de menú y *glass pane*.
protected boolean rootPaneCheckingEnabled	Si es verdadero, llama a *add* y *setLayout* produce una excepción.

Tabla 11.16. Constructores de la clase *JFrame*.

Constructor	Descripción
JFrame()	Crea un nuevo frame
JFrame(Título)	Crea un nuevo frame con el título especificado

Tabla 11.17. *Métodos de la clase JFrame*

Método	Descripción
protected void addImpl(Component como, Object constraints, int indice)	Añade hijos al *contentpane*.
protected JRootPane createRootPane()	Llamado por los métodos del constructor para crear el *root pane* por defecto.
protected void frameInit()	Llamado por los constructores para inicializar la propiedad *JFrame*.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible asociado con este *JFrame*.
Container getContentPane()	Obtiene el objeto *content pane* de este *frame*.
int getDefaultCloseOperation()	Obtiene la operación que tiene lugar cuando el usuario cierra el *frame*.
Component getGlassPane()	Obtiene el objeto *glassPane*.

Método	Descripción
JMenuBar getJMenuBar()	Obtiene la barra de menú.
JRootPane getRootPane()	Obtiene el *root pane.*
protected boolean isrootPaneCheckingEnabled()	Indica si llama a *add* y *setLayout* causa una excepción.
protected String paramString()	Obtiene una representación en cadena de este *JFrame.*
protected void porcessKeyEvent(KeyEvent e)	Procesa los eventos de tecla que ocurren en este componente.
protected void processWindowEvent(WindowEvent e)	Procesa eventos de ventana que ocurren en este componente.
void remove(Component comp)	Elimina el componente de este contenedor.
void setContentPane(Container contentPane)	Obtiene la propiedad *contentPane.*
void setDefaultCloseOperation(int operación)	Fija la operación que tendrá lugar, por defecto, cuando el usuario cierre este *frame.*
void setGlassPane(Componente glassPane)	Fija la propiedad *glassPane.*
void setJMenuBar(JMenuBar menubar)	Establece la barra de menú para este *frame.*
void setLayeredPane(JLayeredPane layeredPane)	Fija la propiedad *layeredPane.*
void setLayout(LayoutManager manager)	Por defecto, se debería fijar el *layout* del *content pane* en su lugar.
protected void setRootPane(JRootPane root)	Fija la propiedad *rootPane.*
protected void setRootPaneCheckingEnabled (boolean enabled)	Determina si llama a *add* y *setLayout* causa una excepción.
void update(Graphics g)	Llama a *paint.*

Ahora, pondremos a funcionar la clase *JFrame*. En este caso, crearemos un objeto *JPanel* (como en el punto anterior), instalaremos el panel en un objeto *JFrame*, y visualizaremos el texto "¡Hola desde Swing!" en el panel. El color de fondo por defecto de *JFrame* es gris, pero haremos que el panel sea blanco fijando su color de fondo. Este es el código:

```
class jpanel extends JPanel
{
    jpanel()
    {
        setBackground(Color.white);
    }

    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("¡Hola desde!", 0, 60);
    }
}
```

A continuación, creamos un objeto de esta nueva clase y lo añadimos al *content pane* de *JFrame* en una aplicación, como sigue:

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

public class app extends JFrame
{
    jpanel j;

    public app()
    {
        super("Aplicación Swing");

        Container contentPane = getContentPane();
        j = new jpanel();
        contentPane.add(j);
    }

    public static void main(String args[])
    {
        final JFrame f = new app();

        f.setBounds(100, 100, 300, 300);
        f.setVisible(true);
    }
}

class jpanel extends JPanel
{
    jpanel()
    {
```

```
        setBackground(Color.white);
    }

    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("¡Hola desde Swing!", 0, 60);
    }
}
```

Ahora, además, hemos usado el método *setBounds* para fijar la posición y tamaño de la ventana *JFrame*. El resultado se muestra en la figura 11.2. Este ejemplo se encuentra en el CD como app.java.

Sin embargo, hay un problema con esta aplicación. Cuando se cierra esta ventana de aplicación haciendo clic en el botón **Cerrar**, desaparece de la pantalla (que es más de lo que una ventana de aplicación AWT haría), pero la aplicación no finaliza. Para remediar esta situación, veamos el siguiente punto.



Figura 11.2. Visualizar una aplicación Swing en una ventana *frame*.

Cerrar ventanas *JFrame*

"Hey", dice el programador novato, "otra vez Java está graciosillo. Cuando intento cerrar una ventana *JFrame* de *Swing*, no hace nada". "Bien", le contesta, "se puede solucionar de varias formas". "Tomemos un café", dice PN.

Cuando se cierra una ventana AWT, por defecto, no pasa nada, pero se pueden gestionar los eventos de ventana y finalizar la aplicación si se quiere. Las ventanas *JFrame* de *Swing*, por otro lado, permiten establecer una operación de cierre por defecto con el método *setDefaultCloseOperation*. Estos son los valores posibles que se le pueden pasar a este método:

- *DO_NOTHING_ON_CLOSE*: El defecto. No pasa nada cuando se cierra la ventana.
- *HIDE_ON_CLOSE*: Esconde la ventana cuando se cierra.
- *DISPOSE_ON_CLOSE*: Se libera de la ventana cuando se cierra. No se puede volver a visualizar la ventana, aunque el objeto todavía está disponible en memoria.

Observe que estas posibilidades sólo tratan con la ventana en sí misma; si se quiere finalizar la aplicación cuando la ventana de aplicación se cierra, todavía tendrá que gestionarse.

A continuación, nos liberamos de la ventana y finalizamos la aplicación con una clase interna adaptadora:

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

public class app extends JFrame
{
    jpanel j;

    public app()
    {
        super("Aplicación Swing");

        Container contentPane = getContentPane();
        j = new jpanel();
        contentPane.add(j);
    }

    public static void main(String args[])
    {
        final JFrame f = new app();

        f.setBounds(100, 100, 300, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

class jpanel extends JPanel
{
```

```
    jpanel()
    {
        setBackground(Color.white);
    }

    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("¡Hola desde Swing!", 0, 60);
    }
}
```

Seleccionar los bordes del componente

"Bien", dice el programador novato, "el gran jefe quiere que nuestros programas sean más decorativos, visualmente más atractivos. ¿Alguna idea?" "Bien", le contesta, "puede usar diferentes tipos de bordes para los componentes en *Swing*. Es bastante fácil". "¿Cómo funciona?" pregunta PN, interesado.

Para crear un borde de un componente, se puede usar la clase *BorderFactory*, que crea bordes de estas clases: *BevelBorder, SoftBevelBorder, EtchedBorder, LineBorder, TitledBorder* y *MatteBorder*. Además se pueden usar las clases *EmptyBorder, CompoundBorder* y *AbstractBorder* para crear los propios bordes.

Este es el diagrama de herencia de la clase *BorderFactory:*

```
java.lang.Object
|____javax.swing.BorderFactory
```

Los métodos de la clase *BorderFactory* se recogen en la tabla 11.18.

La clase *BorderFactory* crea objetos que implementa la interfaz *Border*; los métodos de la interfaz están en la tabla 11.19.

Usar *Insets*

Una parte importante de trabajar con bordes es conocer los *insets*, que indican la distancia que se debe permitir en cada borde para considerarlo en el cálculo.

Los *insets* de un borde se pueden obtener con el método *getBorderInsets* de la interfaz *Border*; este método devuelve un objeto de la clase *Insets*. He aquí el diagrama de herencia de la clase *Insets*:

```
java.lang.Object
|____java.awt.Insets
```

Tabla 11.18. Métodos de la clase *BorderFactory*.

Método	Descripción
static Border createBevelBorder(int tipo*)*	Crea un borde biselado del tipo indicado.
static Border createBevelBorder(int tipo, *Color highlight, Color shadow)*	Crea un borde biselado del tipo, iluminación y sombreado indicados.
static Border createBevelBorder(int tipo, *Color highlightOuter, Color highlightinner, Color shadowOuter, Color shadowInner)*	Crea un borde biselado del tipo indicado, con los colores especificados para las áreas internas y externas de iluminación y sombreado.
static CompoundBorder create CompoundBorder()	Crea un borde compuesto.
static CompoundBorder createCompoundBorder(Border border, Border borde_interior*)*	Crea un borde compuesto, indicando los objetos para los bordes interior y exterior.
static Border createEmptyBorder()	Devuelve un borde vacío que no ocupa espacio. Devuelve un borde vacío.
static Border createEtchedBorder()	Crea un borde como si fuera grabado usando el color de fondo actual del componente para la iluminación y el sombreado.
static Border createEtchedBorder(Color iluminación, *Color* sombra*)*	Crea un borde como si fuera grabado usando los colores de iluminación y sombreado.
static Border createLineBorder(Color color*)*	Devuelve una línea de borde con el color indicado.
static Border createLineBorder(Color color, *int* grosor*)*	Devuelve una línea de borde con el color y grosor indicados.
static Border createLoweredBevelBorder()	Devuelve un borde con un borde grabado más bajo.
static MatteBorder createMatteBorder(int superior, *int* izquierda, *int* fondo, *int* derecha, *Color* color*)*	Devuelve un borde enmarañado usando un color sólido.

Método	Descripción
static MatteBorder createMatteBorder (int superior, *int* izquierda, *int* fondo, *int* derecha, *Icon tileIcon)*	Devuelve un border enmarañado hecho con múltiples azulejos de un icono indicado.
static Border createRaisedBevelBorder()	Devuelve un borde con un borde biselado alzado.
static TitledBorder createTitledBorder (Border borde*)*	Devuelve un borde con un título vacío.
static TitledBorder createTitledBorder (Border borde, *String* titulo)	Añade un título a un borde existente, indicando el texto del título.
static TitledBorder createTitledBorder (Border borde, *String* titulo, *int* Justificacion_del_titulo, *int*Posicion_del_titulo)	Añade un título a un borde, indicando el texto del título con su posición.
static TitledBorder createTitledBorder (Border borde, *String* titulo, *int* Justificacion_del_titulo, *int*Posicion_del_titulo, *Font*Fuente_del_titulo)	Añade un título a un borde, indicando el texto del título con su posición y fuente.
static TitledBorder createTitledBorder (Border borde, *String* titulo, *int* Justificacion_del_titulo, *int*Posicion_del_titulo, *Font* Fuente_del_titulo, *color*Color_del_titulo*)*	Añade un título a un borde, indicando el texto del título con su posición, fuente y color.
static TitledBorder createTitledBorder (String titulo*)*	Crea un nuevo título, indicando el texto y usando las propiedades por defecto.
boolean isBorderOpaque()	Determina si el borde es opaco.
void paintBorder(Component c, Graphics g, int x, int y, int anchura, *int* altura*)*	Pinta el borde del componente especificado.

Tabla 11.19. Métodos de la interfaz *Border*.

Método	Descripción
Insets getBorderInsets(Component c)	Obtiene los *insets* del borde.

Los campos de la clase *Insets* se encuentran en la tabla 11.20, su constructor en la tabla 11.21 y sus métodos en la 11.22.

Tabla 11.20. Campos de la clase *Insets*.

Campo	Descripción
int bottom	*Inset* del fondo.
int left	*Inset* de la izquierda.
int right	*Inset* de la derecha.
int top	*Inset* de la parte superior.

Tabla 11.21. Constructor de la clase *Border*.

Constructor	Descripción
Insets(int superior, *int* izquierda, *int* fondo, *int* derecha*)*	Crea e inicializa un nuevo objeto *Insets*.

Tabla 11.22. Métodos de la clase *Border*.

Método	Descripción
Object clone()	Crea una copia de este objeto.
boolean equals(Object obj)	Verifica si dos objetos son iguales.
String toString()	Obtiene una representación en cadena de este objeto *Insets*.

Veamos un ejemplo. En este caso, se añade un borde a *JPanel* usado en el ejemplo de la *applet* hace algunos apartados. Para hacer eso, sólo usamos el método *setBorder* del panel (que tienen la mayoría de los componentes en *Swing*) y creamos un nuevo borde biselado alzado con el método *createRaisedBevelBorder* de la clase *BorderFactory*:

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

/*
<APPLET
    CODE=borde.class
    WIDTH=300
    HEIGHT=200 >
<A/PPLET>
*/

public class borde extends JApplet
{
```

```
    jpanel j;

    public void init()
    {
        Container contentPane = getContentPane();

        j = new jpanel();
        j.setBorder(BorderFactory.createRaisedBevelBorder());
        contentPane.add(j);
    }
}
```

Después de añadir un nuevo borde al panel, tomaremos los *insets* de este borde para dibujar en su interior el mensaje "¡Hola desde Swing!" sin solaparse con el borde. Para ello, usaremos el método *getBorder* para obtener el borde actual, el método *getBorderInsets* del objeto que soporta la interfaz *Border*, y el miembro de datos *left* de ese objeto para determinar dónde se empieza a pintar el mensaje para que no se solape con el borde:

```
class jpanel extends JPanel
{
    public void paintComponent (Graphics g)
    {
        g.drawString("¡Hola desde Swing!",
        getBorder().getBorderInsets(this).left, 60);
    }
}
```

El resultado se muestra en la figura 11.3. Como se puede ver, el texto aparece en el interior del borde biselado que encierra el panel. Este ejemplo está en el CD como borde.java.

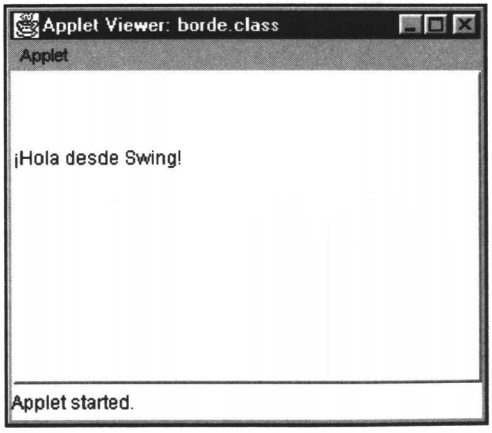


Figura 11.3. Añadir un borde a un objeto *JPanel*.

Establecer la apariencia

El gran jefe (Gj) regresa y dice, "Entiendo que no se puede establecer la apariencia de un programa *Swing* para relacionar varias plataformas de programación". "Es correcto", le dice. BB pregunta, "¿Tiene algo en color caoba?" "Lo siento", le responde.

Por defecto, la apariencia de los programas de *Swing* es la de *Java Metal*, la que Sun ha diseñado independiente de la plataforma. Sin embargo, se puede cambiar usando la clase *UIManager*. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____javax.swing.UIManager
```

El constructor de esta clase se encuentra en la tabla 11.23 y sus métodos en la 11.24.

Para cambiar la apariencia cuando se está poniendo a punto un programa, se puede usar el método *setLookAndFeel* de la clase *UIManager*, pasándole uno de estos argumentos como una cadena:

- *javax.swing.plaf.metal.MetalLookAndFeel*: Apariencia de *Metal*.
- *com.sun.java.swing.plaf.motif.MotifLookAndFeel*: Apariencia de *Motif*.
- *com.sun.java.swing.plaf.windows.WindowsLookAndFeel*: Apariencia de *Windows*.

Tabla 11.23. Constructor de la clase *UIManager*.

Constructor	Descripción
UIManager()	Construye un nuevo objeto *UIManager*.

Tabla 11.24. Métodos de la clase UIManager.

Método	Descripción
static void addAuxiliaryLookAndFeel	Añade un objeto *LookAndFeel* a la lista de apariencias.
static void addpropertyChangeListener (PropertyChangeListener listener)	Añade un *PropertyChangeListener*.
static Object get(Object key)	Obtiene un objeto de la tabla por defecto.

Método	Descripción
static LookAndFeel[] getAuxilizaryLookAndFeels()	Obtiene la lista de apariencias.
static Border getBorder(Object key)	Obtiene un borde de la tabla por defecto.
static Color getColor(Object key)	Obtiene un color para dibujar de la tabla por defecto.
static String getCrossPlatformLookAndFeelClassName()	Obtiene el nombre de la clase *LookAndFeel* que soporta la apariencia independiente de la plataforma por defecto.
static UIDefaults getDefaults()	Obtiene el defecto para esta apariencia.
static Dimension getdimension(Object key)	Obtiene una dimensión de la lista por defecto.
static Font getFont(Object key)	Obtiene una fuente de la lista por defecto.
static Icon getIcon(Object key)	Obtiene un icono de la lista por defecto.
static insets getInsets(Object key)	Obtiene un objeto *Insets* de la lista por defecto.
static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()	Obtiene un *array* de objetos que gestiona la información sobre la implementación *LookAndFeel* instalada.
static int getInt(Object key)	Obtiene un entero de la lista por defecto.
static LookAndFeel getLookAndFeel()	Obtiene la apariencia por defecto actual.
static UIDefaults getLookAndFeelDefaults()	Obtiene los valores por defecto de esta apariencia.
static String getString(Object key)	Obtiene una cadena desde la lista de defectos.
static String getSystemLookAndFeelClassName()	Obtiene el nombre de la clase *LookAndFeel* que implementa los sistemas nativos.

Método	Descripción
static ComponentUI getUI(JComponent target)	Obtiene la apariencia del objeto que dibuja el componente objetivo.
static void installLookAndFeel(String nombre, *String* nombre_de_clase*)*	Crea una nueva apariencia y la añade al *array* actual.
static void installLookAndFeel(UIManager. LookAndFeelInfo info)	Instala la apariencia especificada.
static Object put(Object key, Object valor*)*	Almacena un objeto en los defectos.
static boolean removeAuxiliaryLookAndFeel (LookAndFeel laf)	Elimina un objeto LookAndFeel de la lista de apariencias.
static void removePropertyChangeListener (PropertyChangeListener listener)	Elimina un objeto *PropertyChangeListener.*
static void setInstalledLookAndFeels (UIManager.LookAndFeelInfo[] infos)	Reemplaza el *array* actual de los objetos *LookAndFeelInfo* instalados.
static void setLookAndFeel(LookAndFeel newLookAndFeel)	Fija la apariencia por defecto actual con un objeto *LookAndFeel.*
static void setLookAndFeel(String className)	Fija la apariencia por defecto actual con un nombre de una clase.

Después de cambiar la apariencia del *content pane*, se puede hacer efectiva con el método *updateComponentTreeUI* de la clase *SwingUtilities*, como sigue:

```
SwingUtilities.updateComponentTreeUI(gtContentPane());
```

Este es un ejemplo en el que se añaden muchos controles *Swing* con los que ya hemos trabajado en un *applet*, por lo que se puede ver cómo aparecen con distintas apariencias. Hay tres botones de opción en esta *applet*, etiquetadas como *Metal, Motif* y *Windows,* y cuando se hace clic sobre uno de los botones de opción, la *applet* cambia a la apariencia correspondiente. Este es el código (veremos cómo se usan todos estos controles en los siguientes capítulos):

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

```
import javax.swing.plaf.metal.MetalLookAndFeel;
import com.sun.java.swing.plaf.motif.MotifLookAndFeel;
import com.sun.java.swing.plaf.windows.WindowsLookAndFeel;

/*
<APPLET
    CODE=plaf.class
    WIDTH=210
    HEIGHT=200 >
</APPLET>
*/

public class plaf extends JApplet
{

    JRadioButton b1 = new JRadioButton("Metal"),
    b2 = new JRadioButton("Motif"),
    b3 = new JRadioButton("Windows");

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.add(new jpanel(), BorderLayout.CENTER);
    }

    class jpanel extends JPanel implements ActionListener
    {
        public jpanel()
        {
            add(new JButton("JBotón"));
            add(new JTextField("JCuadro_de_texto"));
            add(new JCheckBox("JCasilla_de_activación"));
            add(new JRadioButton("JBotón_de_opción"));
            add(new JLabel("JEtiqueta"));
            add(new JList(new String[] {
              "JLista Elemento 1", "JLista Elemento 2"",JLista Elemen-
to 3"}));
            add(new JScrollBar(SwingConstants.HORIZONTAL));

            ButtonGroup group = new ButtonGroup();
            group.add(b1);
            group.add(b2);
            group.add(b3);

            b1.addActionListener(this);
            b2.addActionListener(this);
            b3.addActionListener(this);

            add(b1);
            add(b2);
            add(b3);
        }

        public void actionPerformed(ActionEvent e)
        {
```

```
                JRadioButton src = (JRadioButton)e.getSource();

                try {
                    if((JRadioButton)e.getSource() == b1)
                        UIManager.setLookAndFeel(
                          "javax.swing.plaf.metal.MetalLookAndFeel");
                    else if((JRadioButton)e.getSource() == b2)
                        UIManager.setLookAndFeel(
                        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
                    else if((JRadioButton)e.getSource() == b3)
                        UIManager.setLookAndFeel(
                       "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
                }
                catch(Exception ex) {}

                SwingUtilities.updateComponentTreeUI(getContentPane());
            }
        }
    }
```

La figura 11.4 muestra la *applet* con la apariencia *Metal*, la figura 11.15, la de *Motif* y la 11.16, la de *Windows*. Cambiar entre estas apariencias es tan fácil como hacer clic sobre un botón de opción. Este ejemplo está en el CD como plaf.java.

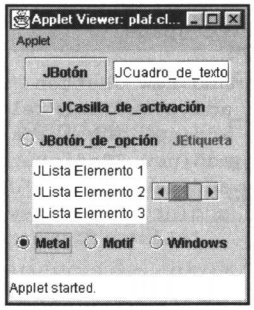


Figura 11.4. Apariencia Java Metal.

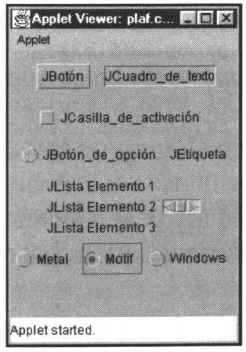


Figura 11.5. Apariencia Motif.

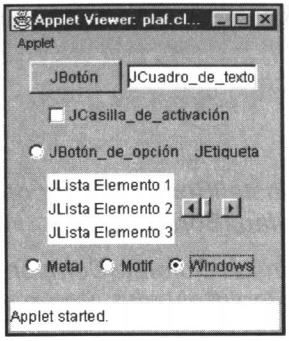


Figura 11.6. Apariencia Windows.

Establecer los componentes para la apariencia

Dice el programador novato: "¿No sería maravilloso si todos los componentes usaran las mismas fuentes y bordes?" "Bien", le contesta, "no estoy seguro de eso, pero se puede hacer en *Swing*". PN pregunta, "¿Usted sabe?"

Se puede establecer la apariencia de los componentes individuales en *Swing* usando la clase *LookAndFeel*. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____javax.swing.LookAndFeel
```

El constructor de la clase *LookAndFeel* se encuentra en la tabla 11.25 y sus métodos en la 11.26.

Tabla 11.25. Constructor de la clase *LookAndFeel*.

Constructor	Descripción
UIDefaults getDefaults()	Lo llama *UIManager.setLookAndFeel* para crear los defectos de la apariencia.
abstract String getDescription()	Obtiene una descripción on line de esta implementación de la apariencia.
abstract String getID()	Obtiene una cadena que identifica esta apariencia.
abstract String getName()	Obtiene una cadena corta que identifica esta apariencia.

Constructor	Descripción
void initialize()	*UIManger.setLookAndFeel* llama a este método antes de la primera llamada a *getDefaults()*.
static void installBorder(JComponent c, String defaultBorderName)	Instala un objeto *Border* por defecto del componente.
static void installColors(JComponent c, String defaultBgName, String defaultFGName)	Instala las propiedades del color de fondo y de primer plano de un componente con los valores de los defectos actuales.
static void installColorAndFont (JCom-ponent c, String defaultBgName, String defaultFontName)	Instala las propiedades de primer plano, fondo y fuente de los componentes con valores por defecto.
abstract boolean isNativeLookAndFeel()	Si es verdadero, ésta es una implementación de la apariencia nativa de la plataforma.
abstract boolean isSupportedLookAndFeel()	Si es verdadero, la plataforma subyacente soporta y/o permite esta apariencia.
static Object makeIcon(Class baseClass, String gifFile)	Crea un *UIDefaults.LazyValue* que crea un *ImageIcon* para el nombre *gifFile* indicado.
static JTextComponent.KeyBinding[] makeKeyBinding(Object[] keyBindingList)	Construye la lista de encuadernaciones clave.
String toString()	Obtiene una cadena que identifica las propiedades de este objeto.
void uninitialize()	*UIManager.setLookAndFeel* llama a este método justo antes de que se instale una nueva apariencia por defecto, reemplazando a ésta.
static void uninstallBorder(JComponent c)	Desinstala un borde por defecto del componente si el borde es una instancia de *UIResource*.

Veamos un ejemplo en el que se da la apariencia de una etiqueta *Swing* a un cuadro de texto, usando los métodos *installBorder* e *installColorsAndFont* de la clase *LookAndFeel*. Observe que todavía no se han visto funcionar estos

controles formalmente; este ejemplo es sólo para indicar qué se puede hacer con estas apariencias. Crearemos una etiqueta *Swing* y una clase extendida desde la clase del cuadro de texto para que parezca una etiqueta. Este es el código:

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

/*
<APPLET
    CODE=plafcomponente.class
    WIDTH=210
    HEIGHT=200 >
</APPLET>
*/

public class plafcomponente extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JNewLabel jnewlabel = new JNewLabel(
                "Esta es una etiqueta falsa.");

        contentPane.setLayout(new FlowLayout());

        contentPane.add(new JLabel("Esta es una etiqueta real."));
        contentPane.add(jnewlabel);
    }
}

class JNewLabel extends JTextField
{
    public JNewLabel(String s)
    {
        super(s);
    }
    public void updateUI()
    {
        super.updateUI();

        setHighlighter(null);
        setEditable(false);

        LookAndFeel.installBorder(this, "Label.border");

        LookAndFeel.installColorsAndFont(this, "Label.background",
            "Label.foreground", "Label.font");
    }
}
```

En la figura 11.7 se muestra el resultado. Como se puede ver en la figura, el cuadro de texto que aparece debajo de la etiqueta tiene la apariencia de una

etiqueta. Esto le indica la potencia de la gestión de apariencias en Swing. Este ejemplo está en el CD como plafcomponente.java.

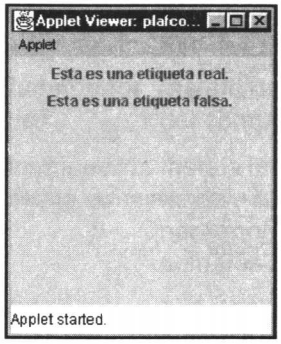


Figura 11.7. Establecer la apariencia de un cuadro de texto como si fuera una etiqueta.

12 *Swing*: Cuadros de texto, botones y casillas de activación

Este capítulo se inicia con un debate sobre los controles *Swing;* en concreto, revisaremos las etiquetas, botones, cuadros de texto, botones *toggle*, casillas de activación y botones de opción en *Swing*. Todos ellos son controles esenciales que ocupan gran parte de los fundamentos de *Swing*. A excepción de los botones *toggle,* todos estos controles le deberían resultar familiares de la programación AWT. Gran parte de la funcionalidad de estos controles *Swing* tiene su correspondencia en la de AWT, y aquí resaltaremos lo que es diferente. (Hay que tener en cuenta que todos los controles *Swing* aparecen de diferentes formas según las diversas apariencias; es importante recordar esto cuando se empiece a programar con ellos).

Etiquetas y cuadros de texto

Las etiquetas son controles básicos *Swing* que sólo visualizan una línea de texto. Sería razonable pensar que las etiquetas soportan varias líneas de texto en *Swing,* pero sólo soportan una. Por otro lado, soportan algo que no tienen

las etiquetas AWT: imágenes. Veremos cómo utilizar la clase *ImageIcon* para añadir imágenes a las etiquetas.

La gestión del texto en los componentes *Swing* es uno de los apartados que veremos más tarde en este libro. Ahora, iniciaremos este tema con los cuadros de texto, viendo cómo funcionan en este capítulo y más adelante.

Botones

En *Swing,* cualquier botón se construye con la clase *AbstractButton,* y en este capítulo veremos esta clase. Como se podía esperar, los botones *Swing* tienen más capacidad que sus correspondientes en AWT, incluyendo la capacidad de visualizar imágenes, usar mnemónicos (combinación de teclas), diseñar un botón como botón por defecto de una ventana y fijar los márgenes y alineación del texto de un botón. Además, a un botón se le pueden asignar múltiples imágenes para gestionar el caso en que el ratón se mueva sobre el botón, y más cosas. Lo veremos a continuación.

Botones *toggle*

Swing introduce los botones *toggle,* que son botones que, cuando se hace clic sobre ellos, permanecen pulsados hasta que volvemos a hacer clic. Los botones *toggle* son como las casillas de activación y botones de opción que parecen botones estándar; de hecho, la clase *JToggleButton,* es la clase base para las clases de estos controles en *Swing*, y además se pueden instanciar objetos de esta clase directamente. Al igual que ocurre con las casillas de activación y botones de opción, se pueden agrupar botones *toggle* y usar imágenes en ellos.

Casillas de activación y botones de opción

AWT tiene casillas de activación y botones de opción, pero los gestiona de diferente forma que *Swing*. En *Swing,* los botones de opción tienen su propia clase (los botones de opción, son casillas de activación en AWT), y con ellos se pueden usar imágenes. De hecho, veremos que cuando se utilizan las imágenes con las casillas de activación y los botones de opción, hay algunos asuntos que considerar.

Con esto finalizamos la visión rápida de lo que contiene este capítulo. Pasemos a la siguiente sección.

Usar etiquetas

El programador novato (PN) aparece y dice, "Estoy creando una nueva *applet Swing* y quiero poner etiquetas a sus controles, por ello estoy empezando con un objeto *Label* y..." "Un momento", le contesta. "Debería estar usando un objeto *JLabel*". PN dice, "Ah".

La clase de las etiquetas peso pesado en *Swing* es *JLabel.* Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____java.awt.Component
          |____java.awt.Container
                    |____javax.swing.JComponent
                              |____javax.swing.JLabel
```

Los constructores de la clase *JLabel* se encuentran en la tabla 12.1 y sus métodos en la tabla 12.2.

Tabla 12.1. Constructores de la clase *JLabel.*

Constructor	Descripción
JLabel()	Construye un objeto *JLabel* sin imagen.
*JLabel(Icon*imagen)	Construye un objeto *JLabel* con la imagen indicada.
*JLabel(Icon*imagen, *int*alineación_horizontal)	Construye un objeto *JLabel* con la imagen y la alineación horizontal especificadas.
*JLabel(String*texto)	Construye un objeto *JLabel* con el texto indicado.
JLabel(String texto, *Icon* icono, *int* alinea -ción_horizontal)	Construye un objeto *JLabel* con el texto, imagen y alineación horizontal indicados.
*JLabel(String*texto, *int*alineación_horizontal)	Construye un objeto *JLabel* con el texto y alineación horizontal indicados.

Tabla 12.2. Métodos de la clase *JLabel*.

Método	Descripción
protected int checkHorizontalKey(int key, String mensaje)	Verifica un valor para las propiedades de alineación horizontal.
protected int checkVerticalKey(int mensaje)	Verifica un valor para la propiedad alineación o la posición de texto verticales.
AccessibleContext getAccessible Context()	Obtiene el contexto accesible.
Icon getDisabledIcon()	Obtiene el valor de la propiedad *disabledIcon* si se ha establecido.
int getDisplayedMnemonic()	Obtiene el código de la tecla que indica un mnemónico.
int getHorizontalAlignment()	Obtiene la alineación de los contenidos de la etiqueta a lo largo del eje x.
int getHorizontalTextPosition()	Obtiene la posición horizontal del texto de la etiqueta, relativo a su imagen.
Icon getIcon()	Obtiene la imagen gráfica (*glyph* o icono) que visualiza la etiqueta.
int getIconTextGap()	Obtiene la cantidad de espacio entre el texto y el icono visualizados en la etiqueta.
Component getLabelFor()	Obtiene el componente del objeto *this*para el que es esta etiqueta.
String getText()	Obtiene la cadena de texto que visualiza la etiqueta.
LabelUI getUI()	Obtiene la apariencia que tiene este componente.
String getUIClassID()	Obtiene una cadena que indica el nombre de la apariencia de la clase de este componente.

Método	Descripción
int getVerticalAlignment()	Obtiene la alineación de los contenidos de esta etiqueta a lo largo del eje y.
int getVerticalTextPosition()	Obtiene la posición vertical del texto de la etiqueta, relativo a su imagen.
protected String paramString()	Obtiene una representación en cadena de este objeto *JLabel.*
*void setDisabledIcon(Icon*icono_deshabilitado)	Fija el icono que se va a visualizar si este objeto*JLabel*está deshabilitado.
void setDisplayedMnemonic(char aChar)	Especifica el mnemónico visualizado como un carácter.
void setDisplayedMnemonic(int key)	Especifica el código de tecla que indica un mnemónico.
*void setHorizontalAlignment(int*alineación)	Fija la alineación de los contenidos de la etiqueta a lo largo del eje x.
*void setHorizontalTextPosition(int*posición_del_texto)	Fija la posición horizontal del texto relativo a su imagen.
void setIcon(Icon icono)	Define el icono que visualizará este componente.
void setIconTextGap(int iconTextGap)	Si el icono y las propiedades del texto están establecidas, esta propiedad define el espacio entre ellas.
void setLabelFor(Component c)	Fija el componente que este objeto está etiquetando.
void setText(String texto)	Define la línea de texto que este componente visualizará.
void setUI(LabelUI ui)	Fija la apariencia del objeto de este componente.
*void setVerticalAlignment(int*alineación)	Fija la alineación de los contenidos de esta etiqueta a lo largo del eje y.

Método	Descripción
void setVerticalTextPosition(int posición_del_texto)	Fija la posición vertical del texto relativo a la imagen.
void updateUI()	Lo llama la clase *UIFactory* cuando la apariencia ha cambiado.

A continuación tenemos un ejemplo básico que muestra cómo se crea una etiqueta *Swing* con el texto "¡Hola desde Swing!":

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

/*
<APPLET
    CODE=etiqueta.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class etiqueta extends JApplet
{
    public etiqueta()
    {
        Container contentPane = getContentPane();

        JLabel jlabel = new JLabel("¡Hola desde Swing!");

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jlabel);
    }
}
```

El resultado de este código se muestra en la figura 12.1; este ejemplo se encuentra en el CD que acompaña a este libro como etiqueta.java.

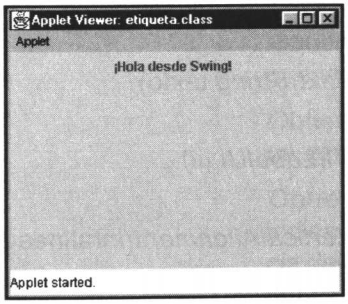


Figura 12.1. Una etiqueta *Swing*.

Hay muchas cosas que se pueden hacer con las etiquetas *Swing*. Por ejemplo, se puede fijar la alineación del texto en una etiqueta, así como utilizar imágenes. Veremos todas las posibilidades en un par de secciones. Para visualizar una imagen en una etiqueta, se necesita un objeto que implemente la interfaz *Icon* (encontrará sus métodos en la tabla 12.3).

Afortunadamente, en *Swing* hay una forma fácil de crear iconos desde las imágenes: se puede usar la clase *ImageIcon*, como se ve en el siguiente punto.

Tabla 12.3. Métodos de la interfaz *Icon*.

Método	Descripción
int getIconHeight()	Obtiene la altura del icono.
int getIconWidth()	Obtiene la anchura del icono.
void paintIcon(Component c, Graphics g, int x, int y)	Dibuja el icono en la posición indicada.

Usar iconos imagen

"Hey", dice el programador novato, "ya eché un vistazo a la interfaz *Icon*. ¿Tengo que pintar las imágenes en los iconos yo mismo cuando utilizo *Swing?*" "Por supuesto que no", le dice. "Se puede utilizar la clase *ImageIcon* para hacer las cosas más fáciles". "Cuénteme todo sobre esto", dice NP.

La clase *ImageIcon* permite crear un icono desde un fichero de imagen para utilizarlo en un control *Swing*. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
                    |____javax.swing.ImageIcon
```

Los constructores de la clase *ImageIcon* se encuentran en la tabla 12.4 y sus métodos en la 12.5.

Tabla 12.4. Constructores de la clase *ImageIcon*.

Constructor	Descripción
ImageIcon()	Construye un icono imagen.
ImageIcon(byte[] imageData)	Construye un icono imagen desde un *array* de bytes.
ImageIcon(byte[] ImageData, String descripción)	Construye un icono imagen desde un *array* de bytes que se leyó de un

Constructor	Descripción
	fichero imagen que contiene un formato de imagen soportado.
ImageIcon(Image imagen)	Construye un icono imagen desde un objeto imagen.
ImageIcon(Image imagen, *String* descripción)	Construye un icono imagen desde la imagen dada.
*ImageIcon(String*nombre_de_fichero)	Construye un icono imagen desde el fichero dado.
*ImageIcon(String*nombre_de_fichero, *String* descripción)	Construye un icono imagen desde el fichero dado.
ImageIcon(URL posición)	Construye un icono imagen desde la URL indicada.
ImageIcon(URL posición, *String* descripción)	Construye un icono imagen desde la URL indicada.

Tabla 12.5. Métodos de la clase *ImageIcon.*

Método	Descripción
String getDescription()	Obtiene la descripción de la imagen.
int getIconHeight()	Obtiene la altura del icono.
int getIconWidth()	Obtiene la anchura del icono.
Image getImage()	Obtiene la imagen del icono.
int getImageLoadStatus()	Obtiene el estado de la operación de carga de imagen.
Imageobserver getImageObserver()	Obtiene el *image observer* de la imagen.
protected void loadImage(Image imagen)	Espera a que se cargue la imagen.
void paintIcon(Component c, Graphics g, int x, int y)	Pinta el icono.
*void setDescription(String*descripción)	Fija la descripción de la imagen.
void setImage(Image imagen)	Fija la imagen visualizada por este icono.
void setImageObserver(ImageObserver observer)	Fija el *image observer* para la imagen.

La clase *ImageIcon* es extremadamente útil en *Swing*, porque muchos componentes *Swing* pueden visualizar iconos. En el siguiente apartado veremos esta clase funcionando.

Usar imágenes en etiquetas

El gran jefe (GJ) aparece y dice: "Necesitamos que las cosas sean un poco más atractivas en nuestros programas, porque ahora, la competencia dentro del campo de GUI es muy fuerte". "De acuerdo", le contesta, "podemos añadir iconos de imágenes a cada etiqueta". "Bien", dice Gj, "su nuevo código era para ayer".

Se pueden utilizar varios constructores *JLabel* para añadir imágenes a las etiquetas, y se puede especificar la alineación de las imágenes y del texto con métodos como estos:

- *setVerticalTextAlignment:* Fija la alineación vertical del texto relativo a la imagen.
- *setHorizontalTextAlignment:* Fija la alineación horizontal del texto relativo a la imagen.
- *setVerticalAlignment:* Fija la alineación vertical de los contenidos de la etiqueta.
- *setHorizontalAlignment:* Fija la alineación horizontal de los contenidos de la etiqueta.

Para especificar las alineaciones se pueden utilizar constantes como estas: *BOTTOM, CENTER, EAST, HORIZONTAL, LEADING, LEFT, NORTH, NORTH_EAST, NORTH_WEST, RIGHT, SOUTH, SOUTH_EAST, SOUTH_WEST, TOP, TRAILING, VERTICAL* y *WEST.*

A continuación tenemos un ejemplo en el que se añade una etiqueta con una imagen y texto a un *applet.* El texto estará centrado debajo de la imagen. Para crear la imagen, usamos la clase *ImageIcon,* pasando al constructor de esa clase el nombre del fichero que gestiona la imagen que queremos usar (que, en este caso, es etiqueta.jpg):

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

/*
<APPLET
    CODE=etiquetaimagen.class
    WIDTH=500
```

```
    HEIGHT=200 >
</APPLET>
*/

public class etiquetaimagen extends JApplet
{

  public void init()
  {
    Container contentPane = getContentPane();

    JLabel jlabel = new JLabel("Etiqueta", new ImageIcon("etiqueta.jpg"),
JLabel.CENTER);
    jlabel.setVerticalTextPosition(JLabel.BOTTOM);
    jlabel.setHorizontalTextPosition(JLabel.CENTER);

    contentPane.add(jlabel);
  }
}
```

El resultado se muestra en la figura 12.2, y este ejemplo se encuentra en el CD como etiquetaimagen.java. Observe lo fácil que es añadir imágenes a las etiquetas en *Swing*. Como es muy sencillo, es muy común.

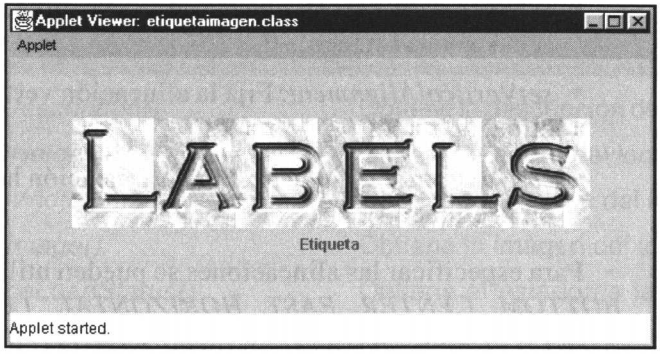


Figura 12.2. Una etiqueta *Swing* con una imagen.

Usar cuadros de texto

"No me lo diga", dice el programador novato, "creo que acertaré el nombre de la clase de *Swing* para los cuadros de texto: *JTextField,* ¿es correcto?" "Exactamente", le sonríe.

Por compatibilidad, los cuadros de texto en *Swing* funcionan como en AWT, con algún valor añadido. Este es el diagrama de herencia de la clase *JTextField:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.text.JTextComponent
                                |____javax.swing.JTextField
```

Más adelante en este libro, veremos más cosas sobre los componentes de texto *Swing,* pero por ahora, introduciremos la clase *JTextField* para que podamos obtener entradas del usuario antes de ese momento. Los constructores de la clase *JTextField* se encuentran en la tabla 12.6 y sus métodos en la 12.7.

Tabla 12.6. Constructores de la clase *JTextField.*

Constructores	Descripción
JTextField()	Construye un nuevo cuadro de texto.
JTextField(Document doc, String texto, *int* columnas)	Construye un nuevo objeto *JTextField* que usa el modelo de almacenamiento y las columnas dados.
JTextField(int columnas)	Construye un nuevo cuadro de texto vacío con el texto indicado.
JTextField(String texto, *int* columnas)	Construye un nuevo cuadro de texto inicializado con el texto y columnas indicados.

Tabla 12.7. Métodos de la clase *JTextField.*

Método	Descripción
void addActionListener(ActionListener l)	Añade el *action listener* para obtener eventos de acción desde este cuadro de texto.
protected Document createDefault-Model()	Construye la implementación, por defecto, del modelo.
portected void fireActionPerformed()	Notifica a los *listeners*que tiene este tipo de evento.
AccessibleContext getAccessible Context()	Obtiene el contexto accesible.
Action[] getActions()	Obtiene la lista de comandos para el editor.
int getColumns()	Obtiene el número de columnas.
protected int getColumnWidth()	Obtiene la anchura de la columna.

Método	Descripción
int getHorizontalAlignment()	Obtiene la alineación horizontal del texto.
BoundedRangeModel getHorizontal-Visibility()	Obtiene la visibilidad del cuadro de texto.
Dimension getPreferredSize()	Obtiene las dimensiones preferidas necesarias para este cuadro de texto.
int getScrollOffset()	Obtiene el *offset* de desplazamiento.
String getUIClassID()	Obtiene la clase ID para un UI.
boolean isValidateRoot()	Vuelve a validar las llamadas que vienen desde el cuadro de texto y que serán gestionadas para validar el nuevo cuadro de texto.
protected String paramString()	Obtiene una representación en cadena de este cuadro de texto.
void positionActionEvent()	Procesa eventos de acción que ocurren en este cuadro de texto enviándolos a cualquier objeto *ActionListener* registrado.
void removeActionListener(Action Listener l)	Elimina el *action listener* indicado.
void scrollRectToVisible(Rectangle r)	Desplaza el campo a la izquierda o derecha.
void setActionCommand(String comando)	Fija la cadena del comando usada para eventos de acción.
void setColumns(int columnas)	Fija el número de columnas.
void setFont(Font f)	Fija la fuente actual.
void setHorizontalAlignment(int alineación)	Fija la alineación horizontal del texto.
void setScrollOffset(int scrollOffset)	Fija el *offset* de desplazamiento.

Este es un ejemplo rápido que sólo visualiza un cuadro de texto con el texto "¡Hola desde Swing!":

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

```
/*
<APPLET
    CODE=cuadrodetexto.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class cuadrodetexto extends JApplet {
    JTextField text = new JTextField(20);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(text);
        text.setText("¡Hola desde Swing!");
    }
}
```

El resultado de este código, cuadrodetexto.java en el CD, se muestra en la figura 12.3.

Esto es sólo el comienzo en la utilización de los componentes de texto en *Swing,* pero nos permite empezar y nos proporciona una forma fácil de visualizar la salida de nuestros programas. Por ejemplo, en los siguientes apartados conectaremos un cuadro de texto a un botón.

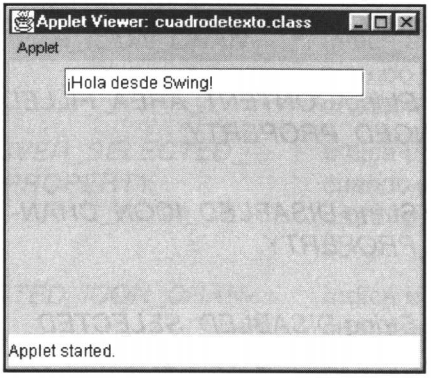


Figura 12.3. Un cuadro de texto *Swing.*

Abstract Button: base de los botones *Swing*

"Pero no quiero usar la clase *AbstractButton"*, dice el programador novato. "Quiero utilizar la clase *JButton.* ¿Por qué tengo que saber nada sobre

AbstractButton?" "En *Swing*, todos los botones se derivan de *AbstractButton*", le dice, "y hay un número tremendo de métodos en esa clase que debería conocer, únicamente como referencia. Realmente, será bueno para usted". "Ah", dice PN.

La clase *AbstractButton* es la base de las clases de botones en Java, y al igual que *JComponent*, proporciona muchos métodos que no podemos ignorar. Este es el diagrama de herencia de *AbstractButton:*

```
java.lang.Object
|____java.awt.Component
          |____java.awt.Container
                    |____javax.swing.JComponent
                              |____javax.swing.text.AbstractButton
```

Los campos de esta clase se encuentran en la tabla 12.8, su constructor en la 12.9 y sus métodos en la 12.10.

Tabla 12.8. Campos de la clase *JAbstractButton.*

Campo	Descripción
protected ActionListener actionListener	*Action listener.*
static String BORDER_PAINTED_CHANGED_PROPERTY	Indica un cambio al borde.
protected ChangeEvent changeEvent	El cambio de evento del botón.
protected ChangeListener changeListener	Los modelos de *listeners* del botón.
static String CONTENT_AREA_FILLED_CHANGED_PROPERTY	Indica un cambio de habilitado a deshabilitado o volver a habilitado.
static String DISABLED_ICON_CHANGED_PROPERTY	Indica un cambio del icono mostrado cuando el botón se ha deshabilitado.
static String DISABLED_SELECTED_ICON_CHANGED_PROPERTY	Indica un cambio del icono mostrado cuando el botón se ha deshabilitado y seleccionado.
static String FOCUS_PAINTED_CHANGED_PROPERTY	Indica un cambio para tener el borde iluminado cuando se tiene o no el foco.
static String HORIZONTAL_ALIGNMENT_CHANGED_PROPERTY	Indica un cambio en la alineación horizontal del botón.

Campo	Descripción
static String HORIZONTAL_TEXT_POSITION_CHANGED_PROPERTY	Indica un cambio en la posición horizontal del texto del botón.
static String ICON_CHANGED_PROPERTY	Indica un cambio en el icono que representa el botón.
protected ItemListener itemListener	El *itemListener* de este botón.
static String MARGIN_CHANGED_PROPERTY	Indica un cambio en los márgenes del botón.
static String MNEMONIC_CHANGED_PROPERTY	Indica un cambio en el nemónico del botón.
protected ButtonModel modelo	Modelo de datos que determina el estado del botón.
static String MODEL_CHANGED_PROPERTY	Indica un cambio en el modelo del botón.
static String PRESSED_ICON_CHANGED_PROPERTY	Indica un cambio en el icono cuando el botón se ha presionado.
static String ROLLOVER_ENABLED_CHANGED_PROPERTY	Indica un cambio en la propriedad "*rolloverenabled*" del botón.
static String ROLLOVER_ICON_CHANGED_PROPERTY	Indica un cambio en el icono cuando el cursor está sobre el botón.
static String ROLLOVER_SELECTED_ICON_CHANGED_PROPERTY	Indica un cambio en el icono cuando el botón está seleccionado y el cursor sobre el botón.
static String SELECTED_ICON_CHANGED_PROPERTY	Indica un cambio en el icono cuando se ha seleccionado el botón.
static String TEXT_CHANGED_PROPERTY	Indica un cambio en el texto del botón.
static String VERTICAL_ALIGNMENT_CHANGED_PROPERTY	Indica un cambio en la alineación vertical del botón.
static String VERTICAL_TEXT_POSITION_CHANGED_PROPERTY	Indica un cambio en la posición vertical del texto del botón.

Tabla 12.9. Constructor de la clase *AbstractButton.*

Constructor	Descripción
AbstractButton()	Construye un nuevo objeto *AbstractButton.*

Tabla 12.10. Métodos de la clase *AbstractButton.*

Método	Descripción
void addActionListener(ActionListener l)	Añade un *action listener.*
void addChangeListener(ChangeListener l)	Añade un *change listener.*
void addItemListener(itemListener l)	Añade un *item listener* a una casilla de activación.
protected int checkHorizontalKey(int properkey, String excepción)	Verifica que *key* es un valor legal para la alineación horizontal.
protected int checkVerticalKey(int key, String excepción)	Verifica que *key* es un valor legal para las propiedades de alineación vertical.
protected ActionListener createActionListener()	Crea un *action listener.*
protected ChangeListener createImplementationChangeListener()	Sobrescribe este método para devolver otro *ChangeListener.*
protected ItemListener createItemListener()	Crea un *item listener.*
void doClick()	Hacer clic en el botón de forma programada.
void doClick(int pressTime)	Hacer clic sobre el botón en un tiempo determinado de forma programada.
protected void fireActionPerformed(Action Event event)	Lanza un evento de acción.
protected void fireItemStateChanged(item Event event)	Lanza un evento de elemento cambiado.
protected void fireStateChanged()	Lanza un evento de estado cambiado.
String getActionCommand()	Obtiene el comando de acción para este botón.

Método	Descripción
Icon getDisabledIcon()	Obtiene el icono deshabilitado.
Icon getDisabledSelectedIcon()	Obtiene el icono seleccionado deshabilitado.
int getHorizontalAlignment()	Obtiene la alineación horizontal del icono y del texto.
int getHorizontalTextPosition()	Obtiene la posición horizontal del texto relativo al icono.
Icon getIcon()	Obtiene el icono por defecto.
String getLabel()	Obsoleto. Reemplazado por *getText()*.
Insets getMargin()	Obtiene el margen entre el borde del botón y la etiqueta.
int getMnemonic()	Obtiene el mnemónico del tecla-do del modelo actual.
ButtonModel getModel()	Obtiene el modelo que representa este botón.
Icon getPressedIcon()	Obtiene el icono presionado para el botón
Icon getrolloverIcon()	Obtiene el icono del botón.
Icon getrolloverSelectedIcon()	Obtiene el icono seleccionado del botón.
Icon getSelectedIcon()	Obtiene el icono seleccionado para el botón.
Object[] getSelectedObjects()	Obtiene un *array* de longitud 1 que contiene la etiqueta o nulo si el botón no está seleccionado.
String getText()	Obtiene el texto del botón.
ButtonUI getUI()	Obtiene el UI actual del botón.
int getVerticalAlignment()	Obtiene la alineación vertical del texto y del icono.
int getVerticalTextPosition()	Obtiene la posición vertical del texto relativa al icono.

Método	Descripción
protected void init(String texto, *Icon* icono)	Para uso interno.
boolean isBorderPainted()	Determina si se debería pintar el borde.
boolean is ContentAreaFilled()	Verifica si se debería rellenar el área de contenido del botón.
boolean isFocusPainted()	Determina si se debería pintar el foco.
boolean isRolloverEnabled()	Verifica si los efectos de *rollover* están activados.
boolean is Selected()	Obtiene el estado del botón.
protected String paramString()	Obtiene una representación en cadena de este objeto *AbstractButton.*
void removeActionListener(ActionListener l)	Elimina un *action listener.*
void removeChangeListener(ChangeListener l)	Elimina un *change listener.*
void removeItemListener(ItemListener l)	Elimina un *item listener.*
void setActionCommand(String action-Command)	Establece el comando de acción.
void setBorderPainted(boolean b)	Fija si se debería pintar el borde.
void setContentAreaFilled(boolean b)	Especifica si el botón debería pintar el *content area.*
void setDisabledIcon(Icon disabledIcon)	Fija el icono deshabilitado para el botón.
void setDisabledSelectedIcon(icon disa-bledSelectedIcon)	Fija el icono seleccionado deshabilitado para el botón.
void setEnabled(boolean b)	Habilita (o deshabilita) el botón.
void setFocusPainted(boolean b)	Fija si se debería pintar el foco.
void setHorizontalAlignment(int alineación)	Fija la alineación horizontal del icono y del texto.
void setHorizontalTextPosition(int textPosition)	Fija la posición horizontal del texto relativa al icono.

Método	Descripción
void setIcon(icon defaulticon)	Fija el icono por defecto del botón.
void setLabel(String etiqueta)	Obsoleto. Reemplazado por *setText(*texto).
void setMargin(Insets m)	Fija el espacio para los márgenes.
void setMnemonic(char mnemonic)	Especifica el valor mnemónico.
void setMnemonic(int mnemonic)	Fija el mnemónico del teclado.
void setMode(ButtonModel nuevo_modelo)	Fija el modelo que representa este botón.
void setPressedIcon(Icon pressedIcon)	Fija el icono presionado.
void setRolloverEnabled(boolean b)	Fija si se deberían habilitar los efectos de *rollover.*
void setRolloverIcon(Icon rolloverIcon)	Fija el icono *rollover* para el botón.
void setRolloverSelectedIcon(Icon rollover-SelectedIcon)	Fija el icono seleccionado *rollover* para el botón.
void setSelected(boolean b)	Fija el estado del botón.
void setSelectedIcon(Icon selectedIcon)	Fija el icono seleccionado para el botón.
void setText(String texto)	Fija el texto del botón.
void setUI(ButtonUI ui)	Fija el UI del botón.
*void setVerticalAlignment(int*alineación)	Fija la alineación vertical del icono y del texto.
*void setVerticalTextPosition(int*posición_del_texto)	Fija la posición vertical del texto relativa al icono.
void updateUI()	Obtiene un nuevo objeto UI desde la clase por defecto *UIFactory.*

Usar botones

El programador novato llega y dice: "Estoy preparado para empezar a trabajar con botones *Swing*. ¿Qué hacen?" "Hmm", le contesta, "¡bastantes cosas! Mejor tome asiento".

La clase *JButton* soporta los botones *Swing*. Este es el diagrama de herencia de esta clase:

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.text.AbstractButton
                                |____javax.swing.JButton
```

Los botones *Swing* permiten hacer más cosas que los botones AWT. Algunas de estas cosas incluyen el uso de *setToolTipText* para añadir un *tooltip* al botón, usar *setMargin* para fijar los *insets* del mismo botón, usar *doClick* para hacer clic en el botón desde el código, añadir imágenes al botón y añadir mnemónicos (combinación de teclas) para el botón. Por supuesto, se pueden hacer cosas estándar de AWT con *JButton*, como usar *setEnabled* para habilitar o deshabilitar el botón, usar *addActionListener* para registrar un *action listener* con el botón y añadir comandos de acción a objetos *JButton* con *setActionCommand*.

Los constructores de la clase *JButton* se encuentran en la tabla 12.11 y sus métodos en la 12.12.

Observe que esta clase se basa en la clase *AbstractButton*, lo que quiere decir que hereda toda su funcionalidad (ver el punto anterior para más detalles).

Tabla 12.11. Constructores de la clase *JButton*.

Constructor	Descripción
JButton()	Construye un botón.
JButton(Icon icono)	Construye un botón con un icono.
JButton(String texto)	Construye un botón con el texto dado.
JButton(String texto, *Icon* icono)	Construye un botón con texto e icono.

Tabla 12.12. Métodos de la clase *JButton*.

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
String getUIClassId()	Obtiene una cadena que indica el nombre de la apariencia.

Método	Descripción
boolean isDefaultButton()	Indica si este botón es el botón por defecto en el *root pane.*
boolean isDefaultCapable()	Indica si este botón es capaz de ser el botón por defecto del *root pane.*
protected String paramString()	Obtiene una representación en cadena de este objeto *JButton.*
void setDefaultCapable(boolean defaultCapable)	Especifica si este botón es capaz de ser el botón por defecto del *root pane.*
void updateUI()	Llamado por la clase *UIFactory* cuando cambia la apariencia.

Veamos un ejemplo en el que está funcionando la clase *JButton.* Aquí se visualiza el mensaje "¡Hola desde Swing!" cuando el usuario hace clic sobre un botón. Este ejemplo, (en el CD botón.java), es muy sencillo, como se puede ver en el siguiente código:

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

/*
<APPLET
    CODE=boton.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class boton extends JApplet {
    JButton button = new JButton("Haga clic aquí");
    JTextField text = new JTextField(20);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);
        contentPane.add(text);

        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event) {
                text.setText("¡Hola desde Swing!");
            }
```

```
        });
    }
}
```

El resultado de este código se muestra en la figura 12.4. Como se puede ver, cuando el usuario hace clic sobre el botón, la *applet* visualiza su mensaje en el cuadro de texto.

Hay mucho más que hacer con los objetos *JButton,* veremos todas las posibilidades en los siguientes apartados.

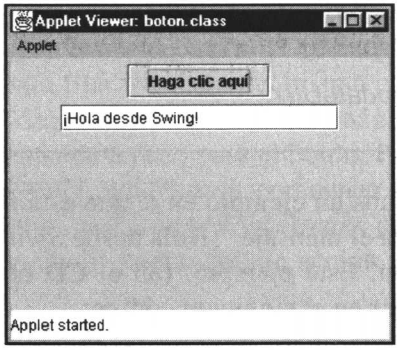


Figura 12.4. Usar un botón *Swing.*

Visualizar imágenes en botones

"Sé que se pueden visualizar imágenes en etiquetas *Swing"*, dice el programador novato, "pero ¿se pueden visualizar en botones?" "No hay ningún problema", le dice, "basta con tener un objeto *Icon*".

Es fácil visualizar imágenes en botones usando la clase *ImageIcon,* ya que varios constructores *JButton* permiten añadir iconos a los botones. Además se puede establecer la alineación del texto y de la imagen usando los métodos *AbstractButton* siguientes:

- *setVerticalTextAlignment:* Fija la alineación vertical del texto relativo a la imagen.
- *setHorizontalTextAlignment:*Fija la alineación horizontal del texto relativo a la imagen.
- *setVerticalAlignment:* Fija la alineación vertical de los contenidos del botón.
- *setHorizontalAlignment:* Fija la alineación horizontal de los contenidos del botón.

Veamos un ejemplo en el que se añade una imagen desde el archivo boton.jpg al ejemplo del botón del apartado anterior:

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

/*
<APPLET
    CODE=botonimagen.class
    WIDTH=600
    HEIGHT=200 >
</APPLET>
*/

public class botonimagen extends JApplet {
    JButton button = new JButton("Botón", new ImageIcon("boton.jpg"));
    JTextField text = new JTextField(20);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);
        contentPane.add(text);

        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event) {
                text.setText("¡Hola desde Swing!");
            }
        });
    }
}
```

El resultado se muestra en la figura 12.5. Como se puede ver ahora, el botón visualiza una gran imagen, haciendo que el programa sea visualmente más interesante. Este ejemplo está en el CD como botonimagen.java.

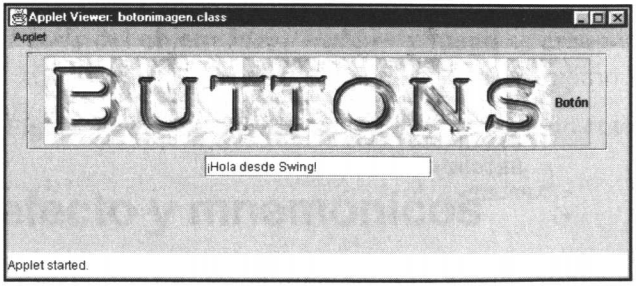


Figura 12.5. Usar un botón *Swing* con una imagen.

Usar imágenes *rollover* y deshabilitadas

El gran jefe está aturullado y dice, "¡Tenemos que alegrar nuestros programas!" "De acuerdo", le dice, "puedo añadir imágenes *rollover* con los botones para que se visualice una imagen cuando el ratón se mueva sobre ellos. Hasta ahora, ¿por qué no hemos animado las cosas?" "Porque", dice el gran jefe, "nuestros competidores han invertido años de trabajo y dinero para hacer que sus programas sean el doble de útiles que los nuestros". "¿No deberíamos hacer nosotros lo mismo?", le pregunta. El GJ dice, "No sea ridículo".

Para los botones se puede establecer un número de iconos. Estas son las posibilidades (iconos *rollover* que aparecen cuando se mueve el ratón sobre el botón):

- Icono normal.
- Icono *rollover.*
- Icono seleccionado *rollover.*
- Icono seleccionado.
- Icono presionado.
- Icono deshabilitado.
- Icono seleccionado deshabilitado.

En el siguiente ejemplo veremos todas estas posibilidades funcionando (en el CD es el archivo botoniconos.java). Para instalar todos estos iconos, usamos un método con el nombre correspondiente: *setIcon, setRolloverIcon, setRolloverSelectedIcon, setSelectedIcon, setPressedIcon, setDisabledIcon* y *setDisabledSelectedIcon.* Aquí tenemos el código (observe que para habilitar los eventos *rollover,* llamamos a *setRolloverEnabled* con un valor *verdadero*):

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

/*
<APPLET
    CODE=botoniconos.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/


public class botoniconos extends JApplet
```

```
{
    public void init()
    {
        Container contentPane = getContentPane();
        Icon normal = new ImageIcon("normal.jpg");
        Icon rollover = new ImageIcon("rollover.jpg");
        Icon pressed = new ImageIcon("pulsado.jpg");
        Icon disabled = new ImageIcon("deshabilitado.jpg");
        Icon selected = new ImageIcon("seleccionado.jpg");
        Icon rolloverSelected = new
            ImageIcon("reseleccionado.jpg");
        Icon disabledSelected = new
            ImageIcon("deseleccionado.jpg");

        JButton jbutton = new JButton();

        jbutton.setRolloverEnabled(true);

        jbutton.setIcon(normal);
        jbutton.setRolloverIcon(rollover);
        jbutton.setRolloverSelectedIcon(rolloverSelected);
        jbutton.setSelectedIcon(selected);
        jbutton.setPressedIcon(pressed);
        jbutton.setDisabledIcon(disabled);
        jbutton.setDisabledSelectedIcon(disabledSelected);

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jbutton);
    }
}
```

El resultado se muestra en la figura 12.6. Cuando el ratón se mueve sobre el botón, la imagen *rollover* está visible.

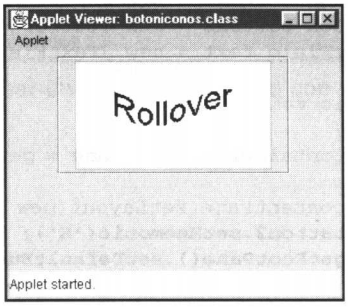


Figura 12.6. Usar imágenes *rollover* y otras en un botón.

Botones por defecto y mnemónicos

Si echa un vistazo a los cuadros de diálogo de su sistema operativo, verá que, generalmente, hay un botón marcado como botón por defecto y ese

botón automáticamente es pulsado si el usuario ejecuta alguna acción con el teclado, como es pulsar la tecla Intro. Con *Swing* se puede poner un botón por defecto usando el método *setDefaultButton.*

Además de hacer botones por defecto, se puede dar a cada botón un mnemónico, que es la combinación de teclas cortas, como se puede ver en los menús. Se subraya una letra (no sensible a mayúsculas y minúsculas) en el texto del botón, y cuando tiene el foco, al pulsar ese carácter el botón se activa. Si el botón no tiene nombre, pulsando la *meta* tecla (por ejemplo, la tecla **Alt** en *Windows)* y el mnemónico del botón lo activa.

Veamos esto en el código. En este caso, se añaden dos botones a un *applet* y se hace que el segundo sea el botón por defecto. Este botón tendrá el literal "Haga clic aquí", y haremos que la letra H sea el mnemónico. Así se haría en el código (observe que damos el foco al *root pane* al final del código para que se puedan interceptar los eventos de tecla y así se active el botón por defecto cuando el usuario pulse la tecla Intro):

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

/*
<APPLET
  CODE = botonpordefecto.class
  WIDTH= 300
  HEIGHT = 200 >
</APPLET>
*/

public class botonpordefecto extends JApplet {
  JButton button1 = new JButton("Haga clic aquí");
  JButton button2 = new JButton("Haga clic aquí");
  JTextField text = new JTextField(20);

  public void init()
  {
      Container contentPane = getContentPane();

      contentPane.setLayout(new FlowLayout());
      button2.setMnemonic('H');
      getRootPane().setDefaultButton(button2);

      contentPane.add(button1);
      contentPane.add(button2);
      contentPane.add(text);
      getRootPane().requestFocus();

      button1.addActionListener(new ActionListener()
      {
            public void actionPerformed(ActionEvent event)
```

```
            {
                text.setText("¡Hola desde Swing!");
            }
        });

        button2.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                text.setText("¡Hola desde Swing!");
            }
        });
    }
}
```

El resultado se muestra en la figura 12.7. Como se puede ver, el segundo botón aparece con un borde resaltado, indicando que es el botón por defecto, y la letra H de su etiqueta está subrayada, indicando que es el mnemónico del botón. Cuando el usuario pulsa la tecla **Intro**, se hace clic sobre el botón por defecto automáticamente, visualizando el mensaje que se muestra en la figura 12.7.

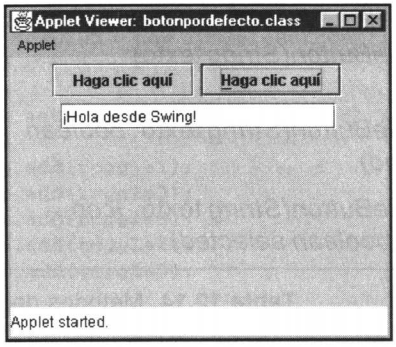


Figura 12.7. Usar un botón por defecto con un mnemónico.

Usar botones *toggle*

El programador novato regresa con una pregunta: "¿Añade *Swing* nuevos tipos de botones a Java?" "Bien", le dice, "sí y no". "Sabía que diría eso", dice PN. "Ahora están los botones *toggle*", le contesta, "pero son realmente la clase base de las casillas de activación y de los botones de opción". "¡Cuénteme más!", dice PN.

Los botones *toggle* son nuevos en *Swing,* y presentan un botón de dos estados (realmente tres estados si se cuenta el estado deshabilitado) que

pueden aparecer como seleccionados o no. Este es el diagrama de *JToggleButton,* la clase de los botones *toggle:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.text.AbstractButton
                                |____javax.swing.JToggleButton
```

Los constructores de esta clase se encuentran en la tabla 12.13 y sus métodos en la 12.14.

Tabla 12.13. Constructores de la clase *JToggleButton.*

Constructor	Descripción
JToggleButton()	Construye un botón *toggle.*
JToggleButton(Icon icono)	Construye un botón *toggle* no seleccionado con la imagen indicada.
JToggleButton(Icon icono, *boolean selected*)	Construye un botón *toggle* con la imagen y estado indicados.
JToggleButton(String texto)	Construye un botón *toggle* no seleccionado con el texto indicado.
JToggleButton(String texto, *boolean selected)*	Construye un botón *toggle* con el texto y estado indicados.
JToggleButton(String texto, *Icon* icono, *boolean selected)*	Construye un botón *toggle* con el texto, imagen y estado indicados.

Tabla 12.14. Métodos de la clase *JToggleButton.*

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
String getUIClassID()	Obtiene una cadena que gestiona el nombre y la apariencia.
protected String paramString()	Obtiene una representación en cadena de este botón *toggle.*
void updateUI()	Llamado por la clase *UIFactory* para indicar que la apariencia ha cambiado.

Usaremos *JToggleButton* en un ejemplo. En él, dibujamos algunos botones *toggle,* la mayoría con iconos y algunos con texto. Así es el código:

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

/*
<APPLET
    CODE=toggle.class
    WIDTH=400
    HEIGHT=400 >
</APPLET>
*/

public class toggle extends JApplet
{
    public toggle()
    {
        Container contentPane = getContentPane();

        Icon icon = new ImageIcon("toggle.jpg");

        JToggleButton toggle1 = new JToggleButton(icon);
        JToggleButton toggle2 = new JToggleButton(icon, true);
        JToggleButton toggle3 = new JToggleButton("¡Haga clic aquí!");
        JToggleButton toggle4 = new JToggleButton("¡Haga clic aquí!",
icon);
        JToggleButton toggle5 = new JToggleButton("¡Haga clic aquí!",icon,
true);

        contentPane.setLayout(new FlowLayout());

        contentPane.add(toggle1);
        contentPane.add(toggle2);
        contentPane.add(toggle3);
        contentPane.add(toggle4);
        contentPane.add(toggle5);
    }
}
```

El resultado de este código se puede ver en la figura 12.8.

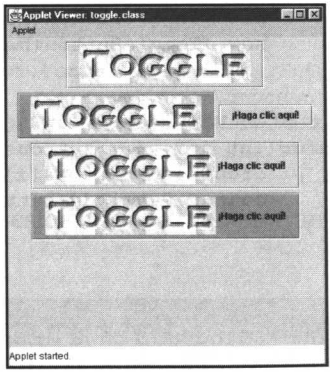


Figura 12.8. Usar botones *toggle*.

Como se puede observar en ella, se fija el estado de un botón *toggle* pasando a su constructor un valor *Verdadero* si se quiere que aparezca seleccionado inicialmente. Este ejemplo está en el CD como *toggle.java.*

Crear grupos de botones *toggle*

El programador novato dice, "Los botones *toggle* son nuevos en *Swing*, ¿hacen algo más que presentar al usuario un estado seleccionado o no seleccionado?" "Claro", le responde, "se pueden agrupar".

Los botones *toggle* se pueden agrupar como cualquier otra clase de botones, con la clase *ButtonGroup:*

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

/*
<APPLET
    CODE=grupotoggle.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/


public class grupotoggle extends JApplet
{
    public grupotoggle()
    {
        Container contentPane = getContentPane();
        ButtonGroup group = new ButtonGroup();

        JToggleButton[] buttons = new JToggleButton[] {
            new JToggleButton(new ImageIcon("toggle.jpg")),
            new JToggleButton(new ImageIcon("toggle.jpg")),
            new JToggleButton(new ImageIcon("toggle.jpg")),
            new JToggleButton(new ImageIcon("toggle.jpg")),
            new JToggleButton(new ImageIcon("toggle.jpg"))
        };

        contentPane.setLayout(new FlowLayout());
        for(int i=0; i < buttons.length; ++i) {
            group.add(buttons[i]);
            contentPane.add(buttons[i]);
        }
    }
}
```

El resultado se muestra en la figura 12.9. Los botones *toggle* están actuando, de hecho, como un grupo; por lo tanto, cuando se hace clic sobre uno, de

ellos cualquiera de los otros botones que estuviera seleccionado se deselecciona automáticamente. Este ejemplo está en el CD como grupotoggle.java.

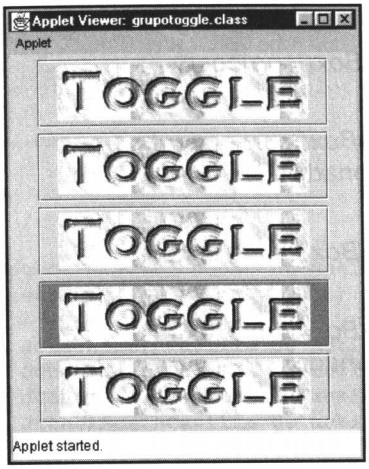


Figura 12.9. Usar botones *toggle* en un grupo.

Usar casillas de activación

El programador novato dice, "Necesito algo para que el usuario pueda seleccionar una opción. Necesito algo para que el usuario pueda seleccionar desde muchas opciones, de hecho. Necesito algo para que el usuario seleccione múltiples opciones de muchas opciones. Necesito..." "Casillas de activación", le responde. "Lo que necesita son casillas de activación". "Cierto", dice PN. La clase *JCheckBox* tiene algunas ventajas sobre la clase AWT *CheckBox,* como visualizar imágenes. Este es el diagrama de herencia de la clase *JCheckBox:*

```
java.lang.Object
|____java.awt.Component
          |____java.awt.Container
                    |____javax.swing.JComponent
                              |____javax.swing.text.AbstractButton
                                        |____javax.swing.JToggleButton
                                                  |____javax.swing.JCheckBox
```

Los constructores de esta clase se encuentran en la tabla 12.15 y sus métodos en la 12.16.



Truco: En futuras versiones de Java, las casillas de activación soportarán texto HTML.

Tabla 12.15. Constructores de la clase *JCheckBox*.

Constructor	Descripción
JCheckBox()	Construye una casilla de activación.
JcheckBox(Icon icono)	Construye una casilla de activación con un icono.
JCheckBox(Icon icono, *boolean* seleccionado)	Construye una casilla de activación con un icono e indica si está seleccionado inicialmente.
JCheckBox(String texto)	Construye una casilla de activación con texto.
JCheckBox(String texto, *boolean* seleccionado)	Construye una casilla de activación con texto e indica si está seleccionado inicialmente.
JCheckBox(String texto, *Icon* icono)	Construye una casilla de activación con el texto e iconos indicados.
JCheckBox(String texto, *Icon* icono, *boolean*seleccionado)	Construye una casilla de activación con texto y un icono e indica si está seleccionado inicialmente.

Tabla 12.16. Métodos de la clase *JCheckBox*.

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
String getUIClassID()	Obtiene una cadena que indica el nombre de la clase de apariencia.
protected String paramString()	Obtiene una representación en cadena de esta casilla de activación.
void updateUI()	Llamado por la clase *UIFactory* para indicar que la apariencia ha cambiado.

Veamos un ejemplo. Aquí, se visualizan cuatro casillas de activación y se indica la que el usuario ha marcado (observe que, como en AWT, se usa *ItemListener* con los objetos *JCheckBox*, no *ActionListener*):

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

/*
```

```
<APPLET
    CODE=casilladeactivacion.class
    WIDTH=340
    HEIGHT=200 >
</APPLET>
*/

public class casilladeactivacion extends JApplet implements ItemListener
{
    JCheckBox check1, check2, check3, check4;
    JTextField text;

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        check1 = new JCheckBox("Casilla de activación 1");
        check2 = new JCheckBox("Casilla de activación 2");
        check3 = new JCheckBox("Casilla de activación 3");
        check4 = new JCheckBox("Casilla de activación 4");

        check1.addItemListener(this);
        check2.addItemListener(this);
        check3.addItemListener(this);
        check4.addItemListener(this);

        contentPane.add(check1);
        contentPane.add(check2);
        contentPane.add(check3);
        contentPane.add(check4);

        text = new JTextField(20);

        contentPane.add(text);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getItemSelectable() == check1) {
            text.setText("Seleccionó la casilla de activación 1.");
        } else if (e.getItemSelectable() == check2) {
            text.setText("Seleccionó la casilla de activación 2.");
        } else if (e.getItemSelectable() == check3) {
            text.setText("Seleccionó la casilla de activación 3.");
        } else if (e.getItemSelectable() == check4) {
            text.setText("Seleccionó la casilla de activación 4.");
        }
    }
}
```

El resultado se puede ver en la figura 12.10. Como se ve, la *applet* devuelve la casilla de activación que se ha seleccionado. Este ejemplo está en el CD como casilladeactivacion.java. Veremos más cosas sobre el uso de las

casillas de activación en *Swing*, en los siguientes apartados, como es el siguiente de los botones de opción.

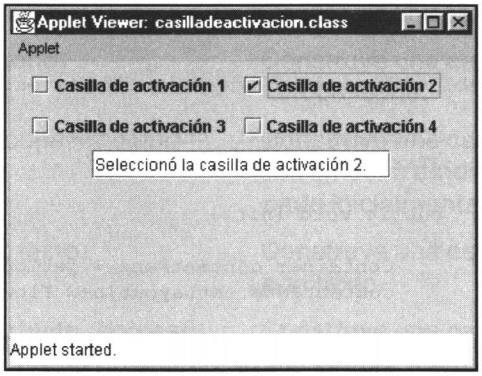


Figura 12.10. Usar casillas de activación.

Usar botones de opción

"De acuerdo", dice el programador novato, "necesito botones de opción en mi código *Swing*. ¿Se hace usando la clase *JCheckBox*?" "No", le contesta. "Aunque se puede usar la clase *CheckBox* para hacer botones de opción en la programación AWT, en *Swing* existe *JRadioButton*". "¡Qué bien!" dice PN.

Este es el diagrama de herencia de la clase *JRadioButton:*

```
java.lang.Object
|____java.awt.Component
        |____java.awt.Container
                |____javax.swing.JComponent
                        |____javax.swing.text.AbstractButton
                                |____javax.swing.JToggleButton
                                        |____javax.swing.JRadioButton
```

A diferencia de la programación AWT, los botones de opción tienen su propia clase en *Swing*, la clase *JRadioButton*. Los constructores de la clase *JRadioButton* se encuentran en la tabla 12.17 y sus métodos en la tabla 12.18.



Truco: en futuras versiones de Java, los botones de opción soportarán texto HTML.

Tabla 12.17. Constructores de la clase *JRadioButton*.

Constructor	Descripción
JRadioButton()	Construye un botón de opción sin texto.
JRadioButton(Icon icono)	Construye un botón de opción con la imagen indicada pero sin texto.
JRadioButton(Icon icono, *boolean* seleccionado)	Construye un botón de opción con la imagen y estado de selección indicados.
JRadioButton(String texto)	Construye un botón de opción con el texto indicado.
JRadioButton(String texto, *boolean* seleccionado)	Construye un botón de opción con el texto y estado de selección indicado.
JRadioButton(String texto, *Icon* icono)	Construye un botón de opción que tiene el texto y la imagen seleccionados.
JRadioButton(String texto, *Icon* icono, *boolean*seleccionado)	Construye un botón de opción que tiene el texto, imagen y estado de selección indicados.

Tabla 12.18. Métodos de la clase *JRadioButton*.

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
String getUICalssID()	Obtiene el nombre de la clase de apariencias.
protected String paramString()	Obtiene una representación en cadena de este botón de opción.
void updateUI()	Llamado por la clase *UIFactory* para indicar que la apariencia ha cambiado.

Veamos un ejemplo. Aquí sólo se visualizan cuatro botones de opción y se agrupan para que sólo se pueda seleccionar uno de ellos al mismo tiempo. Este es el código:

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

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

/*
<APPLET
    CODE=grupobotonesopcion.class
    WIDTH=340
    HEIGHT=200 >
</APPLET>
*/


public class grupobotonesopcion extends JApplet implements ItemListener
{
    JRadioButton radio1, radio2, radio3, radio4;
    ButtonGroup group;
    JTextField text;

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        group = new ButtonGroup();

        radio1 = new JRadioButton("Botón de opción 1");
        radio2 = new JRadioButton("Botón de opción 2");
        radio3 = new JRadioButton("Botón de opción 3");
        radio4 = new JRadioButton("Botón de opción 4");

        group.add(radio1);
        group.add(radio2);
        group.add(radio3);
        group.add(radio4);

        radio1.addItemListener(this);
        radio2.addItemListener(this);
        radio3.addItemListener(this);
        radio4.addItemListener(this);

        contentPane.add(radio1);
        contentPane.add(radio2);
        contentPane.add(radio3);
        contentPane.add(radio4);

        text = new JTextField(20);

        contentPane.add(text);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getItemSelectable() == radio1) {
            text.setText("Seleccionó el botón de opción 1.");
        } else if (e.getItemSelectable() == radio2) {
            text.setText("Seleccionó el botón de opción 2.");
```

```
        } else if (e.getItemSelectable() == radio3) {
            text.setText("Seleccionó el botón de opción 3.");
        } else if (e.getItemSelectable() == radio4) {
            text.setText("Seleccionó el botón de opción 4.");
        }
    }

}
```

El resultado de este código se muestra en la figura 12.11. Todos los botones de opción de la figura actúan juntos, como parte de un grupo. Cuando se hace clic sobre uno, cualquiera que hubiera seleccionado se deselecciona. Este ejemplo está en el CD con el nombre grupobotonesopcion.java.

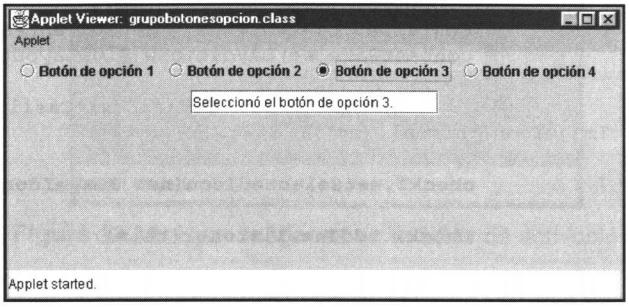


Figura 12.11. Usar botones de opción.

Usar imágenes en casillas de activación y botones de opción

"Uh-oh", dice el programador novato, "de nuevo Java está graciosillo. Añadí una imagen a una casilla de activación y ahora no funciona". Sonríe y le responde, "Eso es porque además hay que añadir una imagen seleccionada a la casilla de activación. De lo contrario, la apariencia de la casilla de activación no cambiará cuando se haga clic sobre ella".

Ahora que ya sabe añadir casillas de activación y botones de opción, observemos un punto importante: si utiliza imágenes en casillas de activación o botones de opción, no hay indicación visual en el control (como la marca de activación) para mostrar si están seleccionados, por lo tanto se debe añadir al control una imagen seleccionada.

Este es un ejemplo que muestra cómo funciona. Aquí, añadimos una imagen seleccionada a una casilla de activación:

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

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

/*
<APPLET
    CODE=imagencasillaactivacion.class
    WIDTH=340
    HEIGHT=200 >
</APPLET>
*/

public class imagencasillaactivacion extends JApplet implements ItemListener
{
    JCheckBox check1;
    JTextField text;

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

            check1 = new JCheckBox("Casilla de activación 1", new
ImageIcon("normal.jpg"));

        check1.setSelectedIcon(new ImageIcon("seleccionado.jpg"));

        check1.addItemListener(this);

        contentPane.add(check1);

        text = new JTextField(20);

        contentPane.add(text);
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getItemSelectable() == check1) {
            text.setText("Seleccionó la casilla de activación 1.");
        }
    }
}
```

El resultado se muestra en la figura 12.12. Cuando el usuario hace clic sobre la casilla de activación, aparece la imagen seleccionada ("*selected*"), como se muestra en la figura. Este ejemplo está en el CD como imagencasilla-activacion.java.

Obtener y fijar el estado de las casillas de activación y de los botones de opción

Es bastante fácil responder a los eventos de las casillas de activación y de los botones de opción, pero a veces se quiere trabajar con estos controles

fuera de los métodos de gestión de eventos. Por ejemplo, el usuario puede seleccionar un número de opciones usando casillas de activación en un cuadro de diálogo que no tienen efecto hasta que se salga del cuadro de diálogo. En ese momento, es necesario determinar qué casillas de activación están seleccionadas.

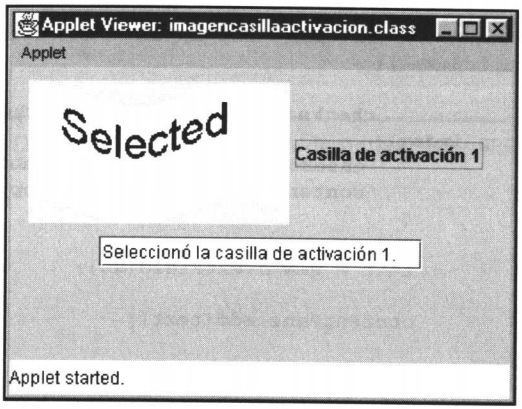


Figura 12.12. Usar imágenes en casillas de activación.

Para determinar si una casilla de activación o botón de opción está seleccionado se puede usar el método *isSelected* y el método *setState* para fijar el estado de una casilla de activación o radio de opción. Este es un ejemplo en el que visualizamos cuatro casillas de activación y listamos las que están marcadas cuando el usuario hace clic sobre ellas.

Empecemos creando un *array* de cuatro casillas de activación y visualizarlos:

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

/*
<APPLET
    CODE=casilladeactivacionseleccionada.class
    WIDTH=500
    HEIGHT=200 >
</APPLET>
*/

public class casilladeactivacionseleccionada extends JApplet implements
ItemListener
{
    JCheckBox checks[];
```

```
    JTextField text;

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        checks = new JCheckBox[4];

          for(int loop_index = 0; loop_index <= checks.length - 1;
loop_index++){

          checks[loop_index] = new JCheckBox("Casilla de activación "
+ loop_index);
           checks[loop_index].addItemListener(this);
           contentPane.add(checks[loop_index]);
        }

        text = new JTextField(40);

        contentPane.add(text);
    }
    .
    .
    .
}
```

Ahora, cuando el usuario hace clic sobre una casilla de activación, recorremos un bucle con todas las casillas de activación, usando el método *isSelected* para ver si están seleccionadas y listar aquellas que lo están:

```
public void itemStateChanged(ItemEvent e)
    {
        String outString = new String("Actualmente seleccionada: ");

          for(int loop_index = 0; loop_index <= checks.length - 1;
loop_index++){
           if(checks[loop_index].isSelected()) {
               outString += " casilla de activación " + loop_index;
           }
        }
        text.setText(outString);
    }
```

El resultado se muestra en la figura 12.13. Como se puede ver, cuando el usuario selecciona las casillas de activación, la *applet* indica las que están seleccionadas.

Este ejemplo está en el CD con el nombre casilladeactivacionseleccionada.java.

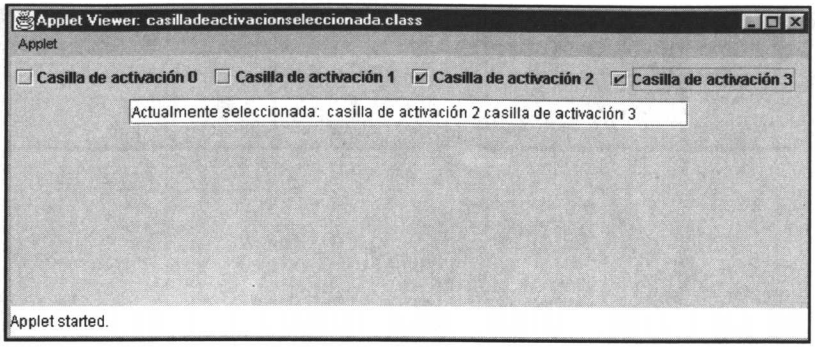


Figura 12.13. Determinar si las casillas de activación están seleccionadas.

13 Swing: viewports, desplazamiento, deslizadores y listas

En este capítulo trataremos algunos temas importantes de la Swing: Viewport, paneles de desplazamiento, deslizadores, barras de desplazamiento y cuadros de lista. Los controles de este capítulo tienen todos una cosa en común: el desplazamiento. El de lista, en concreto, es un control muy importante en la Swing, pero no procesa el desplazamiento por sí mismo, por lo que necesitamos comprender cómo funciona con desplazamiento antes de trabajar con vistas y muchos otros controles de la Swing. Hagamos una breve descripción de los temas de este capítulo antes de profundizar en el código.

Viewports

La clase JViewport es el corazón del desplazamiento en la Swing. Un *viewport* es una ventana dentro de una vista, que visualizará una sección de nuestros datos. Podremos desplazar manualmente los *Viewports*. Utilizando *Viewports*, podemos desplazarnos por los datos visualizados, de forma similar a aquella en que realizamos nosotros mismos el desplazamiento. Examinaremos el uso de la clase JViewport para desplazar imágenes.

Paneles de desplazamiento

Una forma común de implementar el desplazamiento en la Swing es utilizar paneles de desplazamiento, puesto que permiten desplazar componentes. Varios controles de la Swing, como el control JList, implementan la interfaz Scrollable para trabajar con paneles de desplazamiento. De hecho, éstos se utilizan habitualmente con controles JList para crear listas desplazables.

Deslizadores

Otro control desplazable es el control deslizador de la Swing, soportado por la clase JSlider. Los deslizadores son similares a los controles que vemos en los dispositivos de audio que permiten deslizar un mando a lo largo de una pista. De hecho, los deslizadores son como barras de desplazamiento, excepto porque explícitamente utilizamos deslizadores para permitir al usuario seleccionar un valor dentro de un rango continuo.

Puede hacer lo mismo, por supuesto, con barras de desplazamiento, pero los deslizadores se introducen aquí debido a que los usuarios actualmente esperan que las barras de desplazamiento se utilicen para desplazar otros controles, como las áreas de texto.

Barras de desplazamiento

Cada usuario de una UI conoce las barras de desplazamiento, por supuesto, y la Swing las soporta al igual que hacía el AWT. Utilizaremos la clase JScrollBar en este capítulo desplazando texto en un applet. Cuando el cuadro de desplazamiento (también llamado *burbuja* o *marcador*) se mueva, el valor de la barra de desplazamiento cambiará. Puede hacer clic sobre los botones de flecha en los extremos de la barra de desplazamiento para cambiar el valor de la barra de desplazamiento en el *incremento de bloque,* y también puede hacer clic sobre la pista de la barra de desplazamiento para cambiar su valor en su *incremento unitario.*

Listas

Las listas, soportadas por la clase JList en la Swing, son unos controles muy populares, debido a que permiten presentar una lista de elementos senci-

lla de manejar, ocultando una larga lista de elementos al hacer que el cuadro de lista sea desplazable. Mostraremos cómo desplazar largas listas y veremos diversos temas de la Swing. Por ejemplo, puede realizar selecciones múltiples de diversas formas en los cuadros de lista de la Swing. También puede visualizar imágenes, manejar eventos de clics de ratón dobles y triples e incluso aún más. Verá cómo funciona todo esto en este capítulo. También examinaremos cómo implementar un nuevo modelo para listas, así como para renderizadores de celdas, con el fin de manejar la visualización real de cada elemento en la lista de una forma personalizada.

Hasta aquí el resumen de este capítulo. Como puede ver, vendrá mucho más. Es el momento de pasar al primer punto.

Manejo de viewports

"Mmh", dice el programador novato, "quiero mover una imagen bajo control de programa, sin mostrar ninguna barra de desplazamiento al usuario. ¿Existe alguna forma de hacerlo?". "Como es natural", contestamos, "existe; puede utilizar un *viewport*".

Los *viewports* representan ventanas o portales en una vista. Imagine, por ejemplo, que tiene un documento enorme, que no se puede visualizar en la pantalla en un momento dado. Para presentar al usuario una única parte del documento, puede configurar un *viewport* en ese documento y mover el *viewport* a través del documento a medida que lo ve. De hecho, puede tener múltiples *viewports* para permitir que el usuario recorra los datos del modelo de su programa a voluntad. Los *viewport*s están soportados por la clase JViewPort, que es la base del desplazamiento en la Swing. Aquí tenemos el diagrama de herencia para esta clase:

```
java.lang.Object
-java.awt.Component
--java.awt.Container
---javax.swing.JComponent
----javax.swing.JViewPort
```

Encontrará los campos para la clase JViewPort en la tabla 13.1, su constructor en la tabla 13.2 y sus métodos en la tabla 13.3.

Tabla 13.1. Campos de la clase JViewPort.

Campos	Descripción
protected boolean backingStore	Devuelve true cuando el *viewport* mantiene una imagen fuera de pantalla de sus contenidos.

Campos	Descripción
protected Image backingStoreImage	La imagen utilizada para un fondo.
protected boolean isViewSizeSet	Devuelve true cuando se han asignado las dimensiones del *viewport*.
protected Point lastPaintPosition	La última viewPosition pintada.
protected boolean scrollUnderway	El indicador scrollUnderway indica si una operación de desplazamiento está en marcha.

Tabla 13.2. El constructor de la clase JViewPort.

Constructor	Descripción
JViewport()	Construye un objeto JViewport.

Tabla 13.3. Métodos de la clase JViewPort.

Método	Descripción
void addChangeListener(ChangeListener l)	Añade un ChangeListener.
protected void addImpl(Component child, Object constraints, int index)	Asigna el hijo ligero del *viewport*, que puede ser nulo.
protected boolean computeBlit(int dx, int dy, Point blitFrom, Point blitTo, Dimension blitSize, Rectangle blitPaint)	Calcula los parámetros de una operación Blit.
protected LayoutManager createLayoutManager()	Sobrescribe *this* para instalar un distinto gestor de distribución (o nulo) en el constructor.
protected JViewport.ViewListener createViewListener()	Construye un receptor para la vista.
protected JViewport.ViewListener createViewListener()	Notifica a los receptores de un cambio en una propiedad.
protected void fireStateChanged()	Notifica a los receptores de un cambio de estado.
AccessibleContext getAccessibleContext()	Devuelve el contexto accesible.
Dimension getExtentSize()	Devuelve el tamaño de la parte visible de la pista.

Método	Descripción
Insets getInsets()	Obtiene las dimensiones del Inset (borde) como (0,0,0,0). El objeto JViewport no soporta bordes.
Insets getInsets(Insets insets)	Devuelve un objeto Insets conteniendo los valores del Insets del objeto JViewport.
Component getView()	Devuelve el hijo del *viewport* o nulo.
Point getViewPosition()	Devuelve las coordenadas de la vista que aparece en la esquina superior izquierda del *viewport* (0,0 si no existe vista).
Rectangle getViewRect()	Devuelve un rectángulo cuyo origen es getViewPosition() y su tamaño es getExtentSize().
Dimension getViewSize()	Devuelve el tamaño preferido si se ha asignado tamaño; en cualquier otro caso, devuelve el tamaño actual de la vista.
boolean isBackingStoreEnabled()	Devuelve true si el *viewport* mantiene una imagen fuera de pantalla.
boolean isOptimizedDrawingEnabled()	JViewport sobrescribe éste método para devolver falso.
void paint(Graphics g)	Pinta la imagen.
protected String paramString()	Devuelve una representación de cadena de este objeto JViewport.
void remove(Component child)	Elimina el componente hijo ligero del *viewport*.
void removeChangeListener(ChangeListener l)	Elimina un receptor de cambios de la lista.
void repaint(long tm, int x, int y, int w, int h)	Pinta de nuevo la lista.
void reshape(int x, int y, int w, int h)	Asigna los límites de este *viewport*.
void scrollRectToVisible(Rectangle contentRect)	Sobrescrito para desplazar la vista de tal forma que el rectángulo que contiene la vista pase a ser visible.

Método	Descripción
void setBackingStoreEnabled(boolean x)	Si el valor x es true, el *viewport* mantendrá una imagen fuera de pantalla.
void setBorder(Border border)	Asigna un borde.
void setExtentSize(Dimension newExtent)	Asigna el tamaño de la parte visible de la vista utilizando las coordenadas de vista.
void setView(Component view)	Asigna el hijo ligero del *viewport* (su vista).
void setViewPosition(Point p)	Asigna las coordenadas de vista que aparecen en la esquina superior izquierda del *viewport*.
void setViewSize(Dimension newSize)	Asigna las coordenadas de vista que aparecen en la esquina superior izquierda del *viewport* así como el tamaño de la vista.
Dimension toViewCoordinates (Dimension size)	Convierte un tamaño en coordenadas de puntos a coordenadas de vista.
Point toViewCoordinates(Point p)	Convierte un punto en coordenadas de puntos a coordenadas de vista.

Vamos a examinar un ejemplo de Viewport que permite al usuario desplazar el *viewport* con botones y no con barras de desplazamiento. Aquí, añadimos un *viewport* con la imagen a un programa, así como un panel conteniendo botones con los textos desplazar izquierda, desplazar arriba, desplazar abajo y desplazar derecha. El usuario puede desplazar la imagen en el *viewport* haciendo clic con el ratón sobre los botones.

Comencemos creando un nuevo *viewport* y después, un nuevo panel. Añadiremos una etiqueta con una imagen al panel y después añadiremos éste al *viewport*. Cuando el usuario desplace el *viewport*, estará desplazando realmente el panel. Aquí tenemos el mecanismo para crear el *viewport* y añadirlo al programa (observe que para añadir realmente el panel al *viewport*, utilizamos el método setView de la clase JViewPort, que asigna el objeto vista del *viewport*):

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

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

/*
  <APPLET
    CODE=viewport.class
    WIDTH=700
    HEIGHT=200 >
  </APPLET>
*/

public class viewport extends JApplet
{
    public viewport()
    {
        Container contentPane = getContentPane();

        JViewport jviewport = new JViewport();
        JPanel jpanel = new JPanel();

        jpanel.add(new JLabel(new ImageIcon("viewport.jpg")));

        jviewport.setView(jpanel);

        contentPane.add(jviewport, BorderLayout.CENTER);
    }
...
}
```

También necesitamos otro panel que no forme parte del *viewport*, en el que visualizaremos los botones que puede utilizar para el desplazamiento. Llamaremos a éste panel buttonpanel. Observe ahora que pasamos el *viewport* al constructor de esta clase para poder desplazar el *viewport*. Aquí tenemos la forma de almacenar el *viewport* pasado al constructor y añadimos los botones necesarios para el desplazamiento:

```
class buttonpanel extends JPanel implements ActionListener
{
    JViewport jviewport;

    JButton button1 = new JButton("Desplazar a la izquierda");
    JButton button2 = new JButton("Desplazar arriba");
    JButton button3 = new JButton("Desplazar abajo");
    JButton button4 = new JButton("Desplazar a la derecha");

    public buttonpanel(JViewport vport)
    {
        jviewport = vport;

        add(button1);
        add(button2);
        add(button3);
        add(button4);

        button1.addActionListener(this);
```

```
        button2.addActionListener(this);
        button3.addActionListener(this);
        button4.addActionListener(this);
    }
...
}
```

Todo lo que tenemos que hacer es implementar el desplazamiento cuando el usuario haga clic sobre un botón. Para hacerlo, utilizamos el método getViewPosition de la clase JViewPort para obtener la posición actual de la vista. Después, cambiamos esa posición (de acuerdo con el botón sobre el que se haya hecho clic) en 10 puntos y utilizamos el método setViewPosition para asignar la nueva posición del *viewport*:

```
public void actionPerformed(ActionEvent e)
{
    Point position = jviewport.getViewPosition();

    if(e.getSource() == button1) position.x += 10;
    else if(e.getSource() == button2) position.y += 10;
    else if(e.getSource() == button3) position.y -= 10;
    else if(e.getSource() == button4) position.x -= 10;

    jviewport.setViewPosition(position);
}
```

Ahora instalamos un objeto buttonpanel al pie del applet, como sigue:

```
public class viewport extends JApplet
{
    public viewport()
    {
        Container contentPane = getContentPane();

        JViewport jviewport = new JViewport();
        JPanel jpanel = new JPanel();

        jpanel.add(new JLabel(new ImageIcon("viewport.jpg")));

        jviewport.setView(jpanel);

        contentPane.add(jviewport, BorderLayout.CENTER);
      contentPane.add(new buttonpanel(jviewport), BorderLayout.SOUTH);
    }
}
```

El resultado se muestra en la figura 13.1. Como vemos, los botones de desplazamiento aparecen al pie del applet. Cuando el usuario hace clic sobre un botón, la imagen se mueve de forma correspondiente. Este applet es un éxito y se encuentra en el archivo viewport.java del CD que acompaña a este libro.

Normalmente, no utilizaremos siempre la clase JViewPort para manejar nuestro desplazamiento; en su lugar, utilizaremos componentes como paneles de desplazamiento, que examinaremos a continuación.

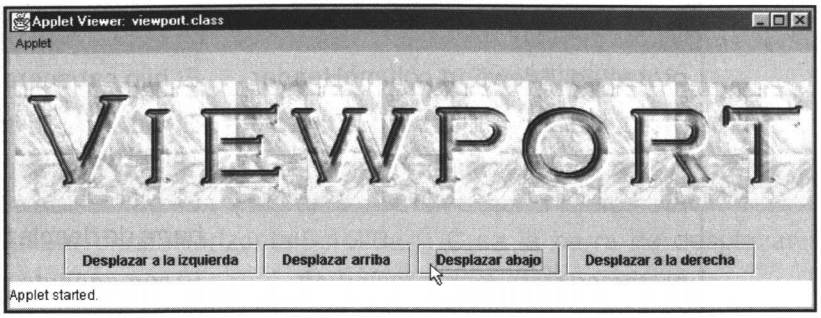


Figura 13.1. Desplazamiento de un viewport.

Creación de paneles de desplazamiento

El programador novato aparece y dice: "Oye, Java ha fallado de nuevo. He creado un nuevo control de lista de la Swing, insertado 40.000 elementos en el mismo y aparecen únicamente dos visibles a la vez y no existen barras de desplazamiento. Esto es un error, voy a crear un informe y...". "Espera", le decimos. "El control JList no implementa desplazamiento por sí mismo, pero lo puedes poner en un panel de desplazamiento". El PN pregunta, "¿De verdad? ¿Es ésa la forma en que *se espera* que funcione?".

La clase JScrollPane es la implementación ligera en la Swing de un panel de desplazamiento y podemos utilizarlo para desplazar otros controles. De hecho, siempre que visualicemos una lista, el mecanismo habitual es visualizarla dentro de un panel de desplazamiento para permitir al usuario desplazar los elementos de la lista.

Gracias a la nueva interfaz Scrollable de la Swing, las operaciones de desplazamiento se coordinan de una forma mucho más próxima al control que se desplaza (las clases JViewPort, JScrollPane y JScrollBar implementan todas esta interfaz). Aquí tenemos el diagrama de herencia para JScroll-Pane:

```
java.lang.Object
-java.awt.Component
--java.awt.Container
---javax.swing.JComponent
---javax.swing.JScrollPane
```

Encontrará los campos para la clase JScrollPane en la tabla 13.4, sus constructores en la tabla 13.5 y sus métodos en la tabla 13.6.

Tabla 13.4. Campos de la clase JScrollPane.

Campos	Descripción
protected JViewport columnHeader	El hijo cabecera de columna.
protected JScrollBar horizontalScroll-Bar	El hijo barra de desplazamiento horizontal del panel de desplazamiento.
protected int horizontalScrollBarPolicy	La política de visualización para la barra de desplazamiento horizontal.
protected Component lowerLeft	El componente a visualizar en la esquina inferior izquierda.
protected Component lowerRight	El componente a visualizar en la esquina inferior derecha.
protected JViewport rowHeader	El componente hijo cabecera de fila.
protected Component upperLeft	El componente a visualizar en la esquina superior izquierda.
protected Component upperRight	El componente a visualizar en la esquina superior derecha.
protected JScrollBar verticalScrollBar	El hijo barra de desplazamiento vertical del panel de desplazamiento.
protected int verticalScrollBarPolicy	La política de visualización para la barra de desplazamiento vertical.
protected JViewport viewport	El hijo panel de desplazamiento del viewport.

Tabla 13.5. Constructores de la clase JScrollPane.

Constructor	Descripción
JScrollPane()	Construye un objeto JScrollPane vacío.
JScrollPane(Component view)	Construye un objeto JScrollPane que visualice los contenidos del componente indicado.
JScrollPane(Component view, int vsbPolicy, int hsbPolicy)	Construye un objeto JScrollPane que visualice el componente vista en un *viewport* cuya posición de

Constructor	Descripción
	vista se pueda controlar con un par de barras de desplazamiento.
JScrollPane(int vsbPolicy, int hsbPolicy)	Construye un objeto JScrollPane vacío con políticas de barras de desplazamiento indicadas.

Tabla 13.6. Métodos de la clase JScrollPane.

Método	Descripción
JScrollBar createHorizontalScrollBar()	Crea la barra de desplazamiento horizontal.
JScrollBar createVerticalScrollBar()	Crea la barra de desplazamiento vertical.
protected JViewport createViewport()	Obtiene un nuevo objeto JScrollPane predeterminado.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible asociado con este objeto JComponent.
JViewport getColumnHeader()	Obtiene la cabecera de columna.
Component getCorner(String key)	Obtiene el componente en la esquina indicada.
JScrollBar getHorizontalScrollBar()	Obtiene la barra de desplazamiento horizontal.
int getHorizontalScrollBarPolicy()	Obtiene el valor de la política de barras desplazamiento horizontal.
JViewport getRowHeader()	Obtiene la cabeza de fila.
ScrollPaneUI getUI()	Obtiene el objeto apariencia que *renderiza* éste componente.
String getUIClassID()	Obtiene la clave utilizada para buscar la clase ScrollPaneUI que proporciona la apariencia para JScrollPane.
JScrollBar getVerticalScrollBar()	Obtiene la barra de desplazamiento vertical.
int getVerticalScrollBarPolicy()	Obtiene el valor de la política para las barras desplazamiento verticales.

Método	Descripción
JViewport getViewport()	Obtiene el objeto JViewPort actual.
Border getViewportBorder()	Obtiene el valor de la propiedad viewportBorder.
Rectangle getViewportBorderBounds()	Obtiene los límites del borde del *viewport*.
boolean isOpaque()	Devuelve true si este componente pinta todos los puntos en su rango.
boolean isValidateRoot()	Comprueba la raíz.
protected String paramString()	Obtiene una representación de cadena de este objeto JScrollPane.
void setColumnHeader(JViewport columnHeader)	Construye un viewport de cabecera de columna, y si es necesario, asigna su vista y después añade el viewport de cabecera de columna al panel de desplazamiento.
void setColumnHeaderView(Component view)	Si existe una cabecera de columna vieja, este método la elimina.
void setCorner(String key, Component corner)	Añade un hijo que aparecerá en una de las esquinas de los paneles desplazamiento (si existe espacio).
void setHorizontalScrollBar(JScrollBar horizontalScrollBar)	Añade la barra de desplazamiento que controla la posición de la barra de vista horizontal del *viewport* al panel de desplazamiento.
void setHorizontalScrollBarPolicy (int policy)	Determina cuándo aparece la barra de desplazamiento horizontal en el panel de desplazamiento.
void setLayout(LayoutManager layout)	Asigna el gestor de distribución para este objeto JScrollPane.
void setRowHeader(JViewport rowHeader)	Asigna la nueva cabecera de fila.
void setRowHeaderView(Component view)	Construye un *viewport* para cabecera de fila, si es necesario, y después añade el viewport para cabecera de fila al panel de desplazamiento.

Método	Descripción
void setUI(ScrollPaneUI ui)	Asigna el objeto que proporciona la apariencia.
void setVerticalScrollBar(JScrollBar verticalScrollBar)	Añade la barra de desplazamiento que controla la posición de vista vertical del *viewport*.
void setVerticalScrollBarPolicy(int policy)	Determina cuándo aparece la barra de desplazamiento vertical en el panel de desplazamiento.
void setViewport(JViewport viewport)	Fuerza la posición de vista del nuevo *viewport*a la posición del cuadrante +x, +y.
void setViewportBorder(Border viewportBorder)	Añade un borde alrededor del *viewport*.
void setViewportView(Component view)	Construye un *viewport*y, si es necesario, asigna su vista.
void updateUI()	Se llama cuando cambia la apariencia predeterminada.

Vamos a examinar un ejemplo en el que añadiremos una distribución de rejilla a los campos de texto de un panel y desplazaremos el panel en un panel de desplazamiento. Comenzaremos creando un objeto JPanel relleno con campos de texto:

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

/*
<APPLET
    CODE=scrollpane.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class scrollpane extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JPanel jpanel = new JPanel();
        jpanel.setLayout(new GridLayout(11, 16));

        for(int outer = 0; outer <= 10; outer++) {
```

```
            for(int inner = 0; inner <= 15; inner++) {
              jpanel.add(new JTextField("Cuadro de texto " + outer +
                  ", " + inner));
            }
        }
    ...
    }
}
```

Ahora, añadiremos este nuevo panel a un panel de desplazamiento. Cuando creemos un objeto panel de desplazamiento, pasaremos el objeto a desplazar y podemos especificar cuándo y dónde deseamos las barras de desplazamiento con estas constantes: HORIZONTAL_SCROLLBAR_ALWAYS, HORIZONTAL_SCROLLBAR_AS_NEEDED, HORIZONTAL_SCROLLBAR_NEVER, VERTICAL_SCROLLBAR, VERTICAL_SCROLLBAR_ALWAYS, VERTICAL_SCROLLBAR_AS_NEEDED, VERTICAL_SCROLLBAR_NEVER.

En este caso, siempre permitiremos que el panel de desplazamiento muestre las barras de desplazamiento:

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

/*
  <APPLET
    CODE=scrollpane.class
    WIDTH=300
    HEIGHT=200 >
  </APPLET>
*/

public class scrollpane extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JPanel jpanel = new JPanel();
        jpanel.setLayout(new GridLayout(11, 16));

        for(int outer = 0; outer <= 10; outer++) {
            for(int inner = 0; inner <= 15; inner++) {
              jpanel.add(new JTextField("Cuadro de texto " + outer +
                  ", " + inner));
            }
        }

        JScrollPane jscrollpane = new JScrollPane(jpanel,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
```

```
            contentPane.add(jscrollpane);
        }
    }
```

El resultado de la adición de un panel de desplazamiento al panel de contenidos del applet se muestra en la figura 13.2. Como muestra la figura, el conjunto completo de la rejilla de campos de texto aparece en el panel de desplazamiento y el usuario puede desplazar esa rejilla con las barras de desplazamiento. Este ejemplo se encuentra en el archivo scrollpane.java del CD.

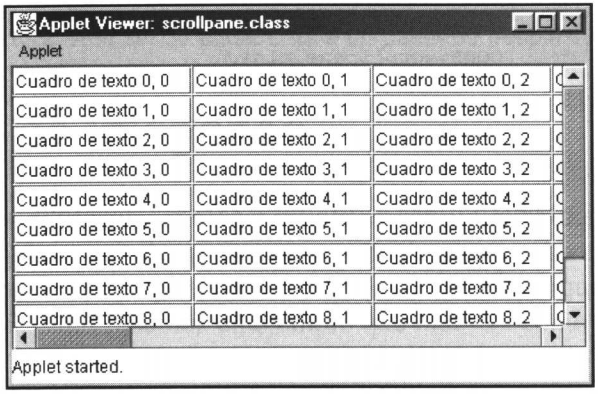


Figura 13.2. Uso de un panel de desplazamiento.

Creación de paneles de desplazamiento con cabeceras y bordes

El zar de la corrección programática aparece y dice: "¿Java no tiene paneles de desplazamiento? Bueno, no tienen una apariencia muy profesional; con controles profesionales, se pueden especificar etiquetas o incluso imágenes para utilizar como cabeceras de filas y columnas". "Sin problema", contestamos. "Se puede hacer en Java e incluso se puede seleccionar el tipo de borde también". "Vaya", dice el ZCP.

Podemos personalizar un panel de desplazamiento añadiendo imágenes de cabecera o texto con los métodos setColumnHeaderView y setRowHeaderView.

También podemos personalizar el borde utilizado en un panel de desplazamiento con el método setViewPortBorder.



Truco: De hecho, incluso podemos personalizar las esquinas de un panel de desplazamiento con el método setCorner.

Como ejemplo, crearemos esta personalización para que trabaje con el ejemplo de panel de desplazamiento que desarrollamos en el tema anterior. En este caso, utilizaremos etiquetas como cabeceras de filas y columnas y añadiremos un borde simple:

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

/*
  <APPLET
    CODE=scrollpane.class
    WIDTH=300
    HEIGHT=200 >
  </APPLET>
*/

public class scrollpane extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JPanel jpanel = new JPanel();
        jpanel.setLayout(new GridLayout(11, 16));

        for(int outer = 0; outer <= 10; outer++) {
            for(int inner = 0; inner <= 15; inner++) {
              jpanel.add(new JTextField("Cuadro de texto " + outer +
                  ", " + inner));
            }
        }

        JScrollPane jscrollpane = new JScrollPane(jpanel,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);

        JLabel jlabel1 = new JLabel("Etiqueta horizontal");
        JLabel jlabel2 = new JLabel("Etiqueta vertical");

        jscrollpane.setColumnHeaderView(jlabel1);
        jscrollpane.setRowHeaderView(jlabel2);
    jscrollpane.setViewportBorder(BorderFactory.createEtchedBorder());

        contentPane.add(jscrollpane);
    }
}
```

La figura 13.3 muestra el resultado de éste código.

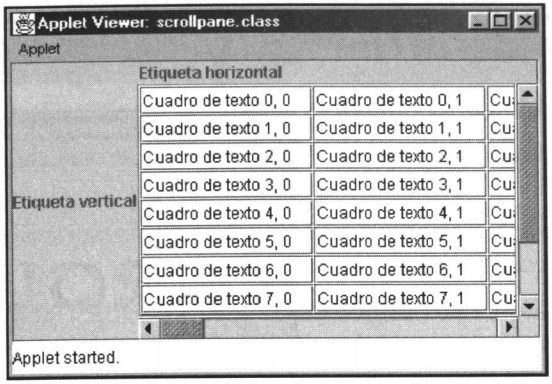


Figura 13.3. Adición de cabeceras a un panel de desplazamiento.

Desplazamiento de imágenes

Los paneles de desplazamiento proporcionan un mecanismo ideal para desplazar imágenes; todo lo que tenemos que hacer es mostrar la imagen en una etiqueta (o en un componente similar) y añadir esa etiqueta al panel de desplazamiento. Aquí tenemos un ejemplo que muestra cómo funciona:

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

/*
  <APPLET
    CODE=scrollpaneimage.class
    WIDTH=300
    HEIGHT=200 >
  </APPLET>
*/


public class scrollpaneimage extends JApplet
{
    public scrollpaneimage()
    {
        Container contentPane = getContentPane();
        JLabel jlabel = new JLabel(new ImageIcon("scrollpane.jpg"));
        JScrollPane jscrollpane = new JScrollPane(jlabel);

        contentPane.add(jscrollpane);
    }
}
```

El resultado se muestra en la figura 13.4. Ahora el usuario puede desplazar una imagen tan grande como desee, para manipular imágenes que no puedan

visualizarse al mismo tiempo. Este ejemplo está en el archivo scrollpaneimage.java en el CD.

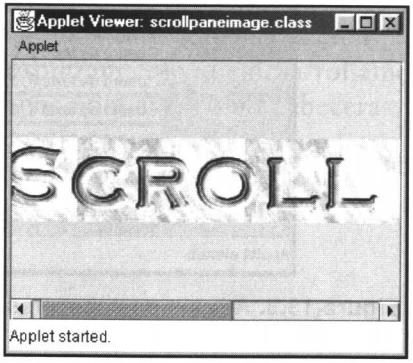


Figura 13.4. Desplazamiento de una imagen.

Creación de deslizadores

El programador novato vuelve y dice, "He visto un nuevo control en varios programas; deslizadores. ¿Cuándo piensas que lo tendremos en Java?". Sonríe y le dice, "Dentro de mucho tiempo".

Podemos utilizar barras de desplazamiento para permitir al usuario seleccionar un valor dentro de un rango continuo, pero los usuarios están más familiarizados con las barras de desplazamiento para otros controles, como las listas.

Por esa razón, Swing añade la clase JSlider para dar soporte a los controles deslizamiento.

Los deslizadores presentan controles al usuario similares a aquellos que vemos en un equipo de audio; están dirigidos claramente a permitir al usuario seleccionar un valor dentro de un rango. Para utilizar los deslizadores, el usuario desplaza un selector con el ratón. Aquí tenemos el diagrama de herencia para la clase JSlider:

```
java.lang.Object
—java.awt.Component
——java.awt.Container
———javax.swing.JComponent
————javax.swing.JSlider
```

Puede encontrar los campos de la clase JSlider en la tabla 13.7, sus constructores en la tabla 13.8 y sus métodos en la tabla 13.9.

Tabla 13.7. Campos de la clase JSlider.

Campos	Descripción
protected ChangeEvent changeEvent	El evento de cambio.
protected ChangeListener changeListener	El receptor de cambio.
protected int majorTickSpacing	El número de valores entre las marcas de incremento mayores.
protected int minorTickSpacing	El número de valores entre las marcas de incremento menores.
protected int orientation	La orientación del deslizador.
protected BoundedRangeModel sliderModel	El modelo de datos, que maneja el valor numérico máximo, valor mínimo y valor para la posición actual del deslizador.
protected boolean snapToTicks	Si vale true, la caja se resuelve a la marca siguiente más cercana a donde el usuario ha situado el cuadro.

Tabla 13.8. Constructores de la clase JSlider.

Constructor	Descripción
JSlider()	Construye un deslizador horizontal con el rango 0 a 100 y un valor inicial de 50.
JSlider(BoundedRangeModel brm)	Construye un deslizador horizontal utilizando el modelo de límites de rango indicado.
JSlider(int orientation)	Construye un deslizador utilizando la orientación indicada con el rango 0 a 100 y un valor inicial de 50.
JSlider(int min, int max)	Construye un deslizador horizontal utilizando los valores mínimo y máximo indicados con un valor inicial de 50.
JSlider(int min, int max, int value)	Construye un deslizador horizontal utilizando los valores mínimo, máximo e inicial indicados.

Constructor	Descripción
JSlider(int orientation, int min, int max, int value)	Construye un deslizador con la orientación indicada y los valores máximo, mínimo e inicial indicados.

Tabla 13.9. Métodos de la clase JSlider.

Método	Descripción
void addChangeListener(ChangeListener l)	Añade un receptor de cambio al deslizador.
protected ChangeListener create ChangeListener()	Sobrescribe este método para devolver su propia implementación del receptor de cambios.
Hashtable createStandardLabels(int increment)	Construye una tabla *hash* que dibujará etiquetas de texto comenzando en el mínimo del deslizador.
Hashtable createStandardLabels(int increment, int start)	Construye una tabla *hash* que dibujará etiquetas de texto comenzando en el punto indicado.
protected void fireStateChanged()	Envía un evento de cambio a cada receptor, cuyo origen es el deslizador.
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
int getExtent()	Obtiene la extensión, que se encuentra en el rango de valores cubierto por el cuadro.
boolean getInverted()	Devuelve true, si el rango de valores mostrado para el deslizador está invertido.
Dictionary getLabelTable()	Obtiene el diccionario para dibujar las etiquetas.
int getMajorTickSpacing()	Obtiene el espaciamiento de marcas mayores.
int getMaximum()	Obtiene el valor máximo soportado por el deslizador.
int getMinimum()	Obtiene el valor mínimo soportado por el deslizador.

Método	Descripción
int getMinorTickSpacing()	Obtiene el espaciamiento de marcas menores.
BoundedRangeModel getModel()	Obtiene el modelo de datos que maneja las tres propiedades fundamentales de los deslizadores: mínimo, máximo y valor.
int getOrientation()	Devuelve la orientación vertical u horizontal del deslizador.
boolean getPaintLabels()	Indica si se tienen que pintar las etiquetas.
boolean getPaintTicks()	Indica si deben pintarse las marcas.
boolean getPaintTrack()	Indica si se debe pintar la pista.
boolean getSnapToTicks()	Devuelve true si el cuadro se resuelve a la siguiente marca más cercana a donde el usuario hubiera posicionado el cuadro.
SliderUI getUI()	Devuelve el objeto IU, que implementa la apariencia para este componente.
String getUIClassID()	Obtiene el nombre de la apariencia que representa este componente.
int getValue()	Obtiene el valor del deslizador.
boolean getValueIsAdjusting()	Devuelve true si se arrastra el cuadro del deslizador.
protected String paramString()	Devuelve una cadena para representar este objeto JSlider.
void removeChangeListener(ChangeListener l)	Elimina un receptor de cambios del deslizador.
void setExtent(int extent)	Asigna el tamaño de rango cubierto por el cuadro.
void setInverted(boolean b)	Pasa un valor JSlider para invertir el rango de valor.
void setLabelTable(Dictionary labels)	Utilizado para especificar qué etiqueta se dibujará en un valor dado.
void setMajorTickSpacing(int n)	Asigna el espaciamiento de marcas mayores.

Método	Descripción
void setMaximum(int maximum)	Asigna la propiedad máximo del modelo.
void setMinimum(int minimum)	Asigna la propiedad mínimo del modelo.
void setMinorTickSpacing(int n)	Asigna el espaciamiento de marcas menores.
void setModel(BoundedRangeModel newModel)	Asigna el modelo que procesa las tres propiedades fundamentales de los deslizadores: mínimo, máximo y valor.
void setOrientation(int orientation)	Asigna la orientación de la barra desplazamiento bien a VERTICAL o HORIZONTAL.
void setPaintLabels(boolean b)	Determina si se pintan etiquetas en el deslizador.
void setPaintTicks(boolean b)	Determina si se pintan marcas en el deslizador.
void setPaintTrack(boolean b)	Determina si se pinta la pista del deslizador.
void setSnapToTicks(boolean b)	Si especificamos true, el cuadro resolverá a la siguiente posición más cercana de las marcas donde se hubiera posicionado el cuadro.
void setUI(SliderUI ui)	Asigna el objeto IU, que implementa la apariencia para este componente.
void setValue(int n)	Asigna valor actual del deslizador.
void setValueIsAdjusting(boolean b)	Asigna la propiedad valueIsAdjusting del modelo.
protected void updateLabelUIs()	Llamada internamente para reemplazar la interfaz de usuario de la etiqueta con las últimas versiones de la clase UIFactory.
void updateUI()	Notificación procedente de la clase UIFactory para indicar que la apariencia ha cambiado.

Vamos a hacer funcionar la clase JSlider en un ejemplo. En este caso, crearemos justamente un deslizador que puede devolver valores desde 0 hasta 100 e informar del valor actual en una barra de estado del applet. Aquí vemos cómo crear un deslizador horizontal con la constante SwingConstants.HORIZONTAL (como podrá suponer, la otra posibilidad es SwingConstants.VERTICAL) con un valor mínimo de 0, un valor máximo de 100 y un valor inicial de 0:

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

/*
  <APPLET
    CODE=slider.class
    WIDTH=300
    HEIGHT=200 >
  </APPLET>
*/

public class slider extends JApplet implements ActionListener,
ChangeListener
{
   JSlider jslider = new JSlider(SwingConstants.HORIZONTAL, 0, 100, 0);
    JButton jbutton = new JButton("Fijada la extensión de texto a 60");

    public void init()
    {
       ...
    }

...
}
```

Ahora, en el método **init**, añadimos este deslizador al panel de contenidos del applet. Utilizaremos *receptores de cambios* con deslizadores y no receptores de ajuste, como hicimos con las barras de desplazamiento. El interfaz ChangeListener tiene únicamente un método, stateChanged, que se muestra en la tabla 13.10. Añadimos un receptor de cambio al deslizador en este applet y después añadimos el deslizador como sigue:

```
public void init()
{
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    jslider.addChangeListener(this);

    contentPane.add(jslider);
}
```

En el método stateChanged, utilizamos los métodos getMinimum, getMaximum y getValue de JSlider para visualizar la configuración del deslizador en la barra de estado:

```
public void stateChanged(ChangeEvent e)
{
    JSlider jslider1 = (JSlider) e.getSource();
    showStatus("Deslizador mínimo: " + jslider1.getMinimum() +
        ", máximo: " + jslider1.getMaximum() +
        ", valor: " + jslider1.getValue() );
}
```

El resultado se muestra en la figura 13.5. Como vemos, el usuario puede mover el selector del deslizador y aparece el nuevo valor. Este ejemplo se encuentra en el archivo slider.java del CD.

Tabla 13.10. El método de la interfaz ChangeListener.

Método	Descripción
void stateChanged(ChangeEvent e)	Invocado cuando el destino del receptor ha cambiado su estado.

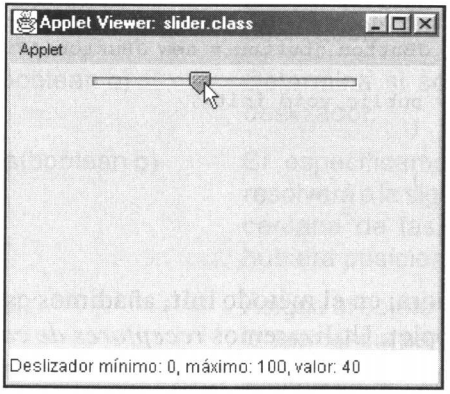


Figura 13.5. Uso de un deslizador.

Relleno de un deslizador

Si está utilizando la apariencia de metal predeterminada de Swing, puede rellenar sus deslizadores, lo que implica que su huella aparecerá rellena desde el origen hasta el selector del deslizador. Para hacerlo, debe asignar la *propiedad cliente* isFilled del JSlider a true, lo que puede hacer como sigue

(observe que estamos añadiendo este código al ejemplo del deslizador a partir del tema anterior):

```
public void init()
{
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    jslider.addChangeListener(this);
    jslider.putClientProperty("JSlider.isFilled", Boolean.TRUE);

    contentPane.add(jslider);
}
```

El resultado aparece en la figura 13.6. Como puede ver, el deslizador de la figura está relleno.

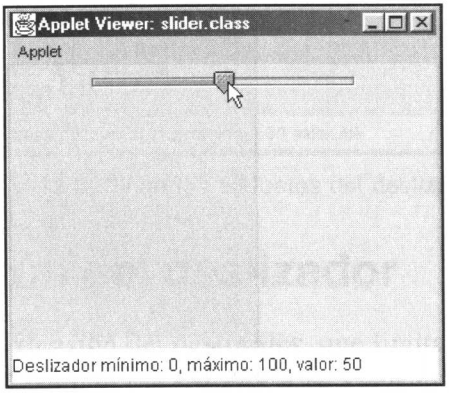


Figura 13.6. Relleno de un deslizador.

Pintar las marcas de un deslizador

"Ah", dice el programador novato, "los usuarios se quejan de los deslizadores en mi programa; dicen que no pueden conseguir lo que quieren utilizando un único deslizador largo".

"Bien", decimos, "puede arreglarlo añadiendo marcas".

Para pintar las marcas en un control de deslizamiento, pasamos un valor de true al método setPaintTicks y después indicamos el desplazamiento mayor y menor que queremos para las marcas, utilizando los métodos setMajorTickSpacing y setMinorTickSpacing, como aquí (observe que estamos añadiendo este código al ejemplo del deslizador existente):

```
public void init()
{
```

```
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        jslider.addChangeListener(this);
        jslider.setPaintTicks(true);
        jslider.setMajorTickSpacing(20);
        jslider.setMinorTickSpacing(10);

        contentPane.add(jslider);
    }
```

El resultado aparece en la figura 13.7. Como se muestran la figura, el deslizador ahora muestra las marcas.

También puede pintar los valores numéricos para las marcas mayores; consulte el siguiente tema para conocer los detalles.

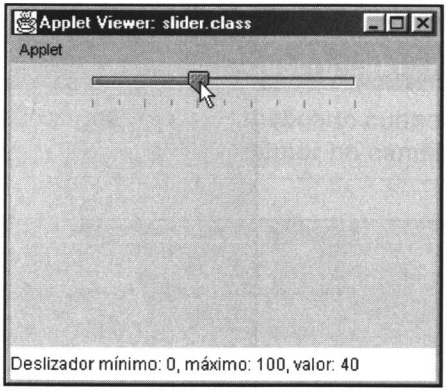


Figura 13.7. Pintar marcas en un deslizador.

Pintar etiquetas en un deslizador

Puede utilizar el método setPaintLabels de la clase JSlider para visualizar el valor numérico de las marcas mayores en el deslizador. Aquí vemos cómo hacerlo añadiendo una línea de código al ejemplo del deslizador existente:

```
public void init()
{
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    jslider.addChangeListener(this);
    jslider.setPaintTicks(true);
    jslider.setPaintLabels(true);
    jslider.setMajorTickSpacing(20);
```

```
        jslider.setMinorTickSpacing(10);

        contentPane.add(jslider);
    }
```

El resultado se muestra en la figura 13.8. Como vemos, se etiquetan las marcas mayores.

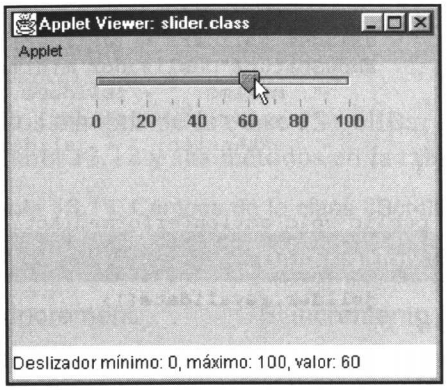


Figura 13.8. Pintar las etiquetas del deslizador.

Ajuste de la extensión del deslizador

Puede ajustar la extensión del deslizador, que limita el valor máximo de éste. Si ajusta la extensión del deslizador y su valor máximo es *máximo*, el valor de deslizador jamás podrá exceder al valor *máximo - extensión*.

Como ejemplo, añadimos un botón al ejemplo deslizador que hemos desarrollado en los últimos temas para permitir que el usuario ajuste la extensión del deslizador a 60:

```
   public class slider extends JApplet implements ActionListener,
ChangeListener
   {
      JSlider jslider = new JSlider(SwingConstants.HORIZONTAL, 0, 100, 0);
      JButton jbutton = new JButton("Fijada la extensión de texto a 60");

       public void init()
       {
           Container contentPane = getContentPane();
           contentPane.setLayout(new FlowLayout());

           jslider.addChangeListener(this);
           jslider.putClientProperty("JSlider.isFilled", Boolean.TRUE);
           jslider.setPaintTicks(true);
           jslider.setPaintLabels(true);
           jslider.setMajorTickSpacing(20);
```

```
        jslider.setMinorTickSpacing(10);

        contentPane.add(jslider);

        jbutton.addActionListener(this);
        contentPane.add(jbutton);
    }

    public void stateChanged(ChangeEvent e)
    {
        JSlider jslider1 = (JSlider) e.getSource();
        showStatus("Deslizador mínimo: " + jslider1.getMinimum() +
            ", máximo: " + jslider1.getMaximum() +
            ", valor: " + jslider1.getValue() +
            ", extensión: " + jslider1.getExtent());
    }

    public void actionPerformed(ActionEvent e)
    {
        jslider.setExtent(60);
        jslider.revalidate();
    }
}
```

Cuando el usuario hace clic sobre el botón, se ajusta la extensión de deslizador a 60 y debido a que su valor máximo es 100, eso significa que su valor no puede superar 40. La figura 13.9 muestra el resultado. Observe que incluso aunque el deslizador esté ajustado a su valor máximo, su valor únicamente es 40.

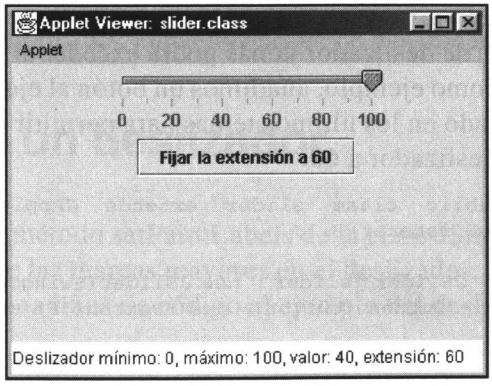


Figura 13.9. Ajuste de la extensión del deslizador.

Creación de barras de desplazamiento

"Bien", dice el programador novato, "quiero permitir al usuario recorrer los datos en mi programa. Quiero dejarle ajustar los valores. Quiero que

pueda navegar por los elementos en una lista larga. Quiero...". "Barras de desplazamiento", respondemos. "Lo que quiere son barras de desplazamiento." "Correcto", dice el PN.

El componente ligero de la Swing para barras de desplazamiento es la clase JScrollBar y su diagrama de herencia es como sigue:

```
java.lang.Object
-java.awt.Component
--java.awt.Container
---javax.swing.JComponent
----javax.swing.JScrollBar
```

Puede encontrar los campos de la clase JScrollBar en la tabla 13.11, sus constructores en la tabla 13.12 y sus métodos en la tabla 13.13.

Tabla 13.11. Campos de la clase JScrollBar.

Campos	Descripción
protected int blockIncrement	El incremento de bloque.
protected BoundedRangeModel model	El modelo que representa los valores actual, mínimo, máximo y extensión de la barra de desplazamiento.
protected int orientation	La orientación de la barra de desplazamiento.
protected int unitIncrement	El incremento unitario.

Tabla 13.12. Constructores de la clase JScrollBar.

Constructor	Descripción
JScrollBar()	Construye una barra de desplazamiento vertical.
JScrollBar(int orientation)	Construye una barra de desplazamiento con la orientación indicada y los siguientes valores iniciales.
JScrollBar(int orientation, int value, int extent, int min, int max)	Construye una barra de desplazamiento con la orientación, valor, extensión, y valores máximo y mínimo indicados.

Tabla 13.13. Métodos de la clase JScrollBar.

Método	Descripción
void addAdjustmentListener(AdjustmentListener l)	Añade un receptor de ajuste.

Método	Descripción
protected void fireAdjustmentValue-Changed(int id, int type, int value)	Dispara un evento de ajuste.
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
int getBlockIncrement()	Utilizado para compatibilidad con versiones anteriores únicamente con java.awt.ScrollBar.
int getBlockIncrement(int direction)	Obtiene la cantidad en que debe cambiarse el valor de la barra de desplazamiento, dado un bloque de petición de cambio.
int getMaximum()	El valor máximo de la barra de desplazamiento es igual al valor máximo menos la extensión.
Dimension getMaximumSize()	Obtiene el tamaño máximo de la barra de desplazamiento.
int getMinimum()	Obtiene el mínimo valor soportado por la barra de desplazamiento.
Dimension getMinimumSize()	Obtiene el tamaño mínimo de la barra de desplazamiento.
BoundedRangeModel getModel()	Obtiene el modelo de datos que procesa las cuatro propiedades fundamentales de la barra de desplazamiento: mínimo, máximo, valor y extensión.
int getOrientation()	Obtiene la orientación del componente (horizontal o vertical).
ScrollBarUI getUI()	Obtiene el delegado que implementa la apariencia de este componente.
String getUIClassID()	Obtiene el nombre de la clase Look-AndFeel para este componente.
int getUnitIncrement()	Utilizado para compatibilidad con versiones previas únicamente para java.awt.ScrollBar.
int getUnitIncrement(int direction)	Obtiene la cantidad que debe cambiar el valor de la barra de despla-

Método	Descripción
	zamiento, dada una unidad de solicitud de cambio.
int getValue()	Obtiene el valor de la barra de desplazamiento.
boolean getValueIsAdjusting()	Devuelve true, si se ha arrastrado el cuadro de la barra de desplazamiento.
int getVisibleAmount()	Obtiene la extensión de la barra de desplazamiento.
protected String paramString()	Obtiene una cadena que representa la barra de desplazamiento
void removeAdjustmentListener (AdjustmentListener l)	Elimina un receptor de AdjustmentEvent.
void setBlockIncrement(int blockIncrement)	Asigna la propiedad blockIncrement.
void setEnabled(boolean x)	Habilita el componente para que la posición del cuadro pueda cambiar.
void setMaximum(int maximum)	Asigna la propiedad máximo del modelo.
void setMinimum(int minimum)	Asigna la propiedad mínimo del modelo.
void setModel(BoundedRangeModel newModel)	Asigna el modelo que procesa las cuatro propiedades fundamentales de la barra de desplazamiento: mínimo, máximo, valor y extensión.
void setOrientation(int orientation)	Asigna la orientación de la barra de desplazamiento a los valores VERTICAL u HORIZONTAL.
void setUnitIncrement(int unitIncrement)	Asigna la propiedad unitIncrement.
Sets the unitIncrement property. void setValue(int value)	Asigna el valor de la barra de desplazamiento.
void setValueIsAdjusting(boolean b)	Asigna la propiedad valueIsAdjusting del modelo.
void setValues(int newValue, int newExtent, int newMin, int newMax)	Asigna las cuatro propiedades BoundedRangeModel.

Método	Descripción
void setVisibleAmount(int extent)	Asigna la propiedad extensión del modelo.
void updateUI()	Sobrescribe JComponent.updateUI.

Aquí tenemos un ejemplo en el que utilizamos una barra de desplazamiento para desplazar texto en un applet.

Para hacerlo, crearemos un nuevo panel que muestre una etiqueta con el texto "¡Hola desde Swing!" en la posición vertical dada por el miembro público de datos denominado **y**. Aquí tenemos la apariencia de la clase del panel:

```
class jpanel extends JPanel
{
    JLabel jlabel = new JLabel("¡Hola desde la Swing!");
    int y = 0;

    jpanel()
    {
        jlabel = new JLabel("¡Hola desde la Swing!");
        add(jlabel);
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        jlabel.setLocation(0, y);
    }

    public void setScrolledPosition(int newposition)
    {
        y = newposition;
    }
}
```

Ahora podemos añadir un objeto de esta nueva clase panel a un applet, así como una barra de desplazamiento vertical en una distribución de borde, como sigue:

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

/*
  <APPLET
    CODE=scrollbar.class
    WIDTH=300
```

```
    HEIGHT=200 >
  </APPLET>
*/

public class scrollbar extends JApplet
{
    private JScrollBar vsb = new JScrollBar(JScrollBar.VERTICAL, 0,
        0, 0, 180);
    private jpanel j = new jpanel();

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.add(j, BorderLayout.CENTER);
        contentPane.add(vsb, BorderLayout.EAST);

        vsb.addAdjustmentListener(new AdjustmentListener()
        {
            public void adjustmentValueChanged(
                                            AdjustmentEvent e) {
                JScrollBar sb = (JScrollBar)e.getSource();
                j.setScrolledPosition(e.getValue());
                j.repaint();
            }
        });
    }
}
```

El resultado se muestra en la figura 13.10. Como muestra la figura, el usuario puede utilizar la barra de desplazamiento para mover el texto arriba y abajo en el panel. Este ejemplo se encuentra en el archivo scrollbar.java del CD.

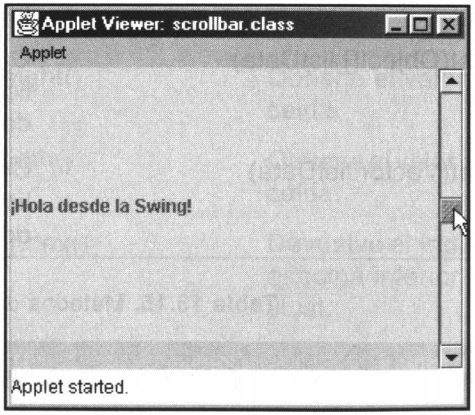


Figura 13.10. Creación y uso de una barra de desplazamiento.

Creación de listas

El gran jefe aparece envuelto en una nube de humo de cigarro y dice, "Ahora tenemos 14.389 productos para que el usuario seleccione. ¿Cómo vamos a visualizarlos todos en un programa?". "Sin problema", respondemos, "utilizamos un control de lista". El gran jefe sonríe y dice, "Aquí tiene los nombres de los productos para introducirlos".

La clase JList de la Swing es un control de lista de poco peso. Aquí tenemos el diagrama de herencia para la clase JList:

```
java.lang.Object
-java.awt.Component
--java.awt.Container
---javax.swing.JComponent
----javax.swing.JList
```

Podemos hacer más con los controles JList de lo que podíamos con los controles List del AWT; por ejemplo, podemos visualizar imágenes en listas Swing. De hecho, examinaremos lo que podemos hacer con las listas Swing desde este punto hasta el final del capítulo. Puede encontrar los constructores de la clase JList en la tabla 13.14 y sus métodos en la tabla 13.15.

Tabla 13.14. Constructores de la clase JList.

Constructor	Descripción
JList()	Construye un objeto JList.
JList(ListModel dataModel)	Construye un objeto JList, que visualizará los elementos en el modelo de datos indicado.
JList(Object[] listData)	Construye un objeto JList que visualice los elementos en la matriz indicada.
JList(Vector listData)	Construye un objeto JList que visualice los elementos en el vector indicado.

Tabla 13.15. Métodos de la clase JList.

Método	Descripción
void addListSelectionListener(ListSelectionListener listener)	Añade un receptor de selección.

Método	Descripción
void addSelectionInterval(int anchor, int lead)	Hace que la selección sea la unión del intervalo indicado con la selección actual.
void clearSelection()	Borra la selección.
protected ListSelectionModel createSelectionModel()	Obtiene una instancia de DefaultListSelectionModel.
void ensureIndexIsVisible(int index)	Desplaza el *viewport* para asegurar que un elemento sea visible.
protected void fireSelectionValueChanged(int firstIndex, int lastIndex, boolean isAdjusting)	Notifica a los receptores de selección de la lista JList que el modelo de selección ha cambiado.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible.
int getAnchorSelectionIndex()	Obtiene el argumento del primer índice a partir de la llamada addSelectionInterval o setSelectionInterval más reciente.
Rectangle getCellBounds(int index1, int index2)	Obtiene los límites del rango de elementos indicado.
ListCellRenderer getCellRenderer()	Obtiene el objeto que renderiza la lista de elementos.
int getFirstVisibleIndex()	Devuelve el índice de la celda en la esquina superior izquierda del objeto JList.
int getFixedCellHeight()	Obtiene el valor de altura fija de la celda.
int getFixedCellWidth()	Obtiene el valor de anchura fija de la celda.
int getLastVisibleIndex()	Devuelve el índice de la celda en la esquina inferior derecha del objeto JList.
int getLeadSelectionIndex()	Obtiene el argumento del segundo índice a partir de la llamada addSelectionInterval o setSelectionInterval más reciente.

Método	Descripción
int getMaxSelectionIndex()	Obtiene el índice máximo de la celda seleccionada.
int getMinSelectionIndex()	Obtiene el índice mínimo de la celda seleccionada.
ListModel getModel()	Obtiene el modelo de datos que contiene la lista de elementos de datos.
Dimension getPreferredScrollable ViewportSize()	Calcula el tamaño del *viewport* necesario para visualizar visibleRow-Count filas.
Object getPrototypeCellValue()	Obtiene el ancho de celda de la celda prototipo (una celda utilizada para el cálculo de anchos de celdas).
int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)	Obtiene la cantidad de incremento de bloque.
boolean getScrollableTracksViewportHeight()	Obtiene la altura de la pista del *viewport*.
boolean getScrollableTracksViewportWidth()	Obtiene el ancho de la pista del *viewport*.
int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)	Devuelve el tamaño del tipo de letra de la lista
int getSelectedIndex()	Devuelve el índice del primer seleccionado.
int[] getSelectedIndices()	Devuelve una matriz con todos los índices seleccionados en orden creciente.
Object getSelectedValue()	Devuelve el primer valor seleccionado o nulo, si no hay selección.
Object[] getSelectedValues()	Devuelve una matriz de valores para las celdas seleccionadas.
Color getSelectionBackground()	Obtiene el color de fondo de las celdas seleccionadas.
Color getSelectionForeground()	Obtiene el color de primer plano.

Método	Descripción
int getSelectionMode()	Obtiene si están permitidas o no las listas simples o múltiples.
ListSelectionModel getSelection-Model()	Obtiene el valor del modelo de selección actual.
ListUI getUI()	Obtiene la apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la clase UIFactory que genera la apariencia para este componente.
boolean getValueIsAdjusting()	Obtiene el valor de la propiedad valueIsAdjusting del modelo de datos.
int getVisibleRowCount()	Devuelve el número preferido de filas visibles.
Point indexToLocation(int index)	Obtiene el origen del elemento indicado en coordenadas de JList. Devuelve nulo si el índice no es válido.
boolean isSelectedIndex(int index)	Devuelve true si el índice seleccionado es el indicado.
boolean isSelectionEmpty()	Devuelve true si no hay nada seleccionado. Es un método conveniente que únicamente delega en el modelo de selección.
int locationToIndex(Point location)	Convierte en un punto en coordenadas JList al índice de la celda en esa localización.
protected String paramString()	Obtiene una representación de cadena de este objeto JList.
void removeListSelectionListener (ListSelectionListener listener)	Elimina al receptor de la lista que recibe notificación cada vez que ocurra un cambio en la selección.
void removeSelectionInterval (int index0, int index1)	Establece la selección para que sea la diferencia como conjunto del intervalo indicado y la selección actual.

Método	Descripción
void setCellRenderer (ListCellRenderer cellRenderer)	Asigna el delegado que se utiliza para pintar cada celda de la lista.
void setFixedCellHeight(int height)	Define la altura de todas las celdas de la lista.
void setFixedCellWidth(int width)	Define la anchura de todas las celdas de la lista.
void setListData(Object[] listData)	Construye un modelo de lista a partir de una matriz de objetos y después aplica el setModel al modelo.
void setListData(Vector listData)	Construye un modelo de lista a partir de un vector y después aplica setModel.
void setModel(ListModel model)	Asigna al modelo que representa los contenidos de la lista y limpia la selección.
void setPrototypeCellValue (Object prototypeCellValue)	Utilizado para calcular fixedCell-Width y fixedCellHeight.
void setSelectedIndex(int index)	Selecciona una única celda.
void setSelectedIndices(int[] indices)	Selecciona un conjunto de celdas.
void setSelectedValue(Object anObject, boolean shouldScroll)	Selecciona el objeto indicado de la lista.
void setSelectionBackground (Color selectionBackground)	Asigna el color de fondo para las celdas seleccionadas.
void setSelectionForeground (Color selectionForeground)	Asigna al color de primer plano para las celdas seleccionadas.
void setSelectionInterval(int anchor, int lead)	Selecciona el intervalo indicado
void setSelectionMode (int selectionMode)	Determina si están permitidas las selecciones simples o múltiples.
void setSelectionModel (ListSelectionModel selectionModel)	Asigna al modelo de selección para la lista a la implementación no nula del modelo de selección ListSelec-tionModel.
void setUI(ListUI ui)	Asigna el objeto que renderiza la apariencia de este componente.

Método	Descripción
void setValueIsAdjusting(boolean b)	Asigna la propiedad isAdjusting del modelo de datos a true.
void setVisibleRowCount (int visibleRowCount)	Asigna el número preferido de filas en la lista que puede visualizarse sin una barra de desplazamiento.
void updateUI()	Asigna la propiedad IU con el objeto ListUI a partir de la clase UIFactory predeterminada.

Observe que puede obtener el índice del elemento seleccionado actualmente con el método getSelectedIndex de la clase JList, selecciones múltiples con getSelectedIndices, el elemento seleccionado actualmente con getSelectedValue y los elementos seleccionados en el momento con getSelectedValues.



Truco: Una cosa a tener en cuenta es que los datos de la lista los mantiene realmente su modelo, por lo que si deseamos acceder a un elemento que no está seleccionado, podemos utilizar el método getModel de la clase JList para obtener el modelo y después el método getElementAt del modelo para conseguir el elemento real.

Haremos funcionar la clase JList en éste y los temas siguientes. Para comenzar, crearemos un ejemplo sencillo JList que únicamente visualice 12 elementos e informe sobre cuál de ellos se ha hecho clic. Comenzaremos creando un control de lista que contendrá esos elementos:

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

/*
   <APPLET
     CODE=list.class
     WIDTH=300
     HEIGHT=200 >
   </APPLET>
*/

public class list extends JApplet implements ListSelectionListener
{
    JList jlist;
    public void init()
```

```
    {
        Container contentPane = getContentPane();
        String[] items = new String[12];

        for(int loop_index = 0; loop_index <= 11; loop_index++) {
            items[loop_index] = "Elemento " + loop_index;
        }

        jlist = new JList(items);
        ...
    }

    ...
}
```

Después, añadimos este control de lista a un panel de desplazamiento para permitir el desplazamiento, lo configuraremos para que visualice cinco filas y añadiremos un receptor de selección para él:

```
public void init()
{
    Container contentPane = getContentPane();
    String[] items = new String[12];

    for(int loop_index = 0; loop_index <= 11; loop_index++) {
        items[loop_index] = "Elemento " + loop_index;
    }

    jlist = new JList(items);
    JScrollPane jscrollpane = new JScrollPane(jlist);
    jlist.setVisibleRowCount(5);
    jlist.addListSelectionListener(this);

    contentPane.setLayout(new FlowLayout());
    contentPane.add(jscrollpane);
}
```

Ahora, en el método valueChanged, podemos utilizar el método getSelectedIndex para obtener el elemento sobre el que ha hecho clic el usuario y visualizar el número del elemento en la barra de estado:

```
public void valueChanged(ListSelectionEvent e)
{
    String outString = "Seleccionó el elemento: ";
    outString += jlist.getSelectedIndex();
    showStatus(outString);
}
```

Eso es todo lo que hace falta. Ahora, cuando el usuario hace clic sobre un elemento de la lista, el applet visualiza sobre qué elemento se ha hecho clic, como se muestra en la figura 13.11. Este ejemplo se encuentra en el archivo list.java del CD.

Observe, sin embargo, que existe más de una forma de seleccionar elementos en una lista; puede realizar selecciones múltiples también sobre listas. Examine el siguiente tema para conocer los detalles.

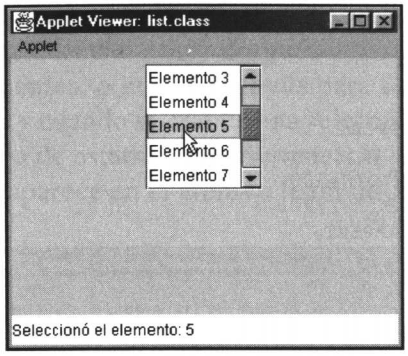


Figura 13.11. Gestión de selecciones en un control de lista.

Gestión de selecciones múltiples

Aparece el especialista de soporte a productos y comenta: "A los usuarios no les agrada el nuevo programa. Quieren reordenar doce productos a la vez, pero el control de lista del programa únicamente les permite ordenar uno". "¿AH sí?", preguntamos. "Por tanto el gran jefe va a ser muy infeliz", dice el ESP. "Bien", decimos, "configuraremos un control de lista de selección múltiple".

Modos de selección de lista

Por defecto, existen tres modos de selección para los objetos JList y podemos asignar el deseado con el método setSelectionMode. Aquí tenemos los constantes que podemos pasar a este método y lo que significan:

- SINGLE_SELECTION. Selección simple.
- SINGLE_INTERVAL_SELECTION. Selección en un intervalo. El usuario puede seleccionar un único intervalo con elementos.
- MULTIPLE_INTERVAL_SELECTION. Intervalos de selección múltiple.

Crearemos un control de lista de selección con intervalo múltiple y mostraremos cómo determinar qué elementos están seleccionados cuando el usuario

realiza una selección. Para comenzar, vamos a crear un control de lista como en el tema anterior, convirtiéndolo en un control de lista de selección de intervalo múltiple. Aquí tenemos el código:

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

/*
  <APPLET
   CODE=listmultiple.class
   WIDTH=300
   HEIGHT=200 >
  </APPLET>
*/

public class listmultiple extends JApplet implements ListSelectionListener
{
    JList jlist;

    public void init()
    {
        Container contentPane = getContentPane();
        String[] items = new String[12];

        for(int loop_index = 0; loop_index <= 11; loop_index++) {
            items[loop_index] = "Elemento " + loop_index;
        }

        jlist = new JList(items);
        JScrollPane jscrollpane = new JScrollPane(jlist);
        jlist.setVisibleRowCount(5);
    jlist.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        jlist.addListSelectionListener(this);

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jscrollpane);
    }

    ...
}
```

Ahora, cuando el usuario realiza una selección, podemos determinar qué elementos fueron seleccionados en el método valueChanged, utilizando el método getSelectedIndices, que devuelve una matriz con los índices seleccionados. El resultado final es que podemos informar de estos índices en la barra de estado del applet, como sigue:

```
    public void valueChanged(ListSelectionEvent e)
    {
        int[] indexes = jlist.getSelectedIndices();
        String outString = "Su selección:";
```

```
        for(int loop_index = 0; loop_index < indexes.length; loop_index++){
            outString += " elemento " + indexes[loop_index];
        }
        showStatus(outString);
    }
```

El resultado se muestra en la figura 13.12. El usuario puede hacer clic sobre un elemento para seleccionarlo y después utilizar la tecla **Control** para seleccionar otros elementos, o la tecla **Mayús** para seleccionar un intervalo de elementos. Siempre y cuando se realice una selección, el applet informa de la selección en su barra de estado, como se muestra en la figura 13.12. Este ejemplo es un éxito y aparece en el archivo listmultiple.java del CD.

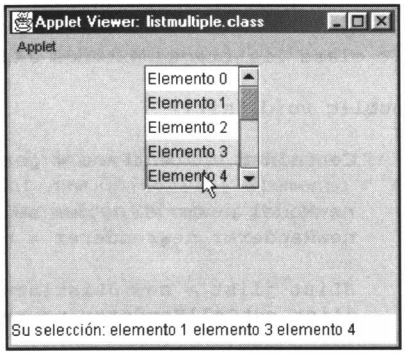


Figura 13.12. Gestión de selecciones múltiples en un control de lista.

Visualización de imágenes en listas

"Tenemos que mejorar las cosas", dice el gran jefe, "la competencia está siendo demasiado dura". "Bien", preguntamos, "¿qué tal si añadimos más funcionalidades?". "Eso cuesta dinero", ruge el gran jefe. "Ajá", decimos, "supongo que podríamos visualizar imágenes en los controles de lista". "Perfecto", dice el gran jefe, "hagámoslo".

Hacer que un control de lista de la Swing visualice imágenes no es tan fácil como añadir imágenes a una etiqueta de la Swing. En particular, tenemos que crear nuestro propio modelo y nuestra propia clase de volcado para el control de lista y lo haremos aquí. El resultado que buscamos se muestra en la figura 13.13. Este ejemplo se encuentra en el archivo listimages.java del CD.

Para gestionar imágenes en un control de lista, crearemos un nuevo modelo de control de lista, newModel, y un nuevo control de lista para volcar celdas, newRenderer. Para instalar el nuevo modelo, que contendrá la cadena de cada elemento de lista y el icono de imagen para cada uno de ellos,

podemos pasar un objeto de la clase newModel al constructor JList. Para instalar el nuevo volcador, podemos utilizar el método setCellRenderer. Aquí tenemos el código:

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

/*
  <APPLET
   CODE=listimages.class
   WIDTH=300
   HEIGHT=300 >
  </APPLET>
*/

public class listimages extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        newModel newmodel = new newModel();
        newRenderer newrenderer = new newRenderer();

        JList jlist = new JList(newmodel);
        jlist.setCellRenderer(newrenderer);
        jlist.setVisibleRowCount(6);

        contentPane.add(new JScrollPane(jlist));
    }
}
```

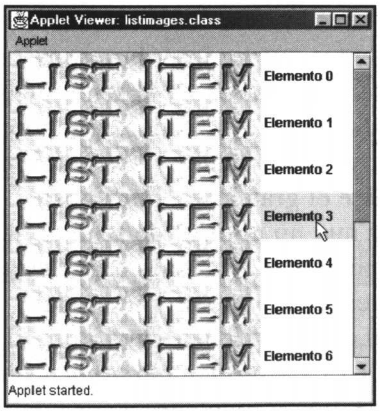


Figura 13.13. Adición de imágenes a un control de lista.

Esto instala el nuevo modelo y *renderizador*; lo único queda realmente es crear las clases correspondientes y lo haremos en los dos temas siguientes.

Creación de un modelo de lista personalizado

La creación de un nuevo modelo para una lista no es difícil; todo lo que tenemos que hacer es almacenar nuestros datos utilizando el método addElement. En este caso, crearemos el nuevo modelo para el ejemplo de lista del tema anterior, que visualizará imágenes y texto en un control de lista.

En este caso, solamente haremos que cada elemento del modelo sea una matriz de dos elementos; el primer elemento de la matriz contendrá el texto para el elemento de lista y el segundo elemento de la matriz contendrá el icono de imagen para el elemento de lista. Aquí tenemos el código:

```
class newModel extends DefaultListModel
{
    public newModel()
    {
        for(int loop_index = 0; loop_index <= 12; loop_index++) {
            addElement(new Object[] {"Elemento " + loop_index,
                new ImageIcon("item.jpg")});
        }
    }
}
```

Esto es todo. Ahora hemos creado un nuevo modelo para un control de lista. Para dibujar realmente el icono de la imagen, sin embargo, tenemos que procesar nosotros mismos el volcador de la celda. Lo haremos en el tema siguiente.

Creación de un *renderizador* personalizado para celdas de lista

Cada elemento de lista es realmente de clase JLabel, por lo que podemos utilizar los métodos de JLabel, como setText y setIcon, para visualizar imágenes en las celdas de la lista.

Aquí, lo haremos mediante la creación de un nuevo *renderizador* de celda que dibujará los elementos en la lista que ya hemos desarrollado en los dos temas anteriores. Para crear un *renderizador* de celdas, tenemos que implementar la interfaz ListCellRenderer, que tiene únicamente un método: getListCellRendererComponent. Este objeto recibe el objeto a volcar (que en este caso es la matriz bidimensional que contiene el texto e icono del elemento) y devuelve un *renderizador* de celda. Aquí tenemos el resultado del

código (observe que también indicamos si está seleccionado un elemento asignando su fondo):

```
class newRenderer extends JLabel implements ListCellRenderer
{
    public newRenderer()
    {
        setOpaque(true);
    }

    public Component getListCellRendererComponent(
        JList jlist, Object obj, int index, boolean isSelected,
        boolean focus)
    {
        newModel model = (newModel)jlist.getModel();

        setText((String)((Object[])obj)[0]);
        setIcon((Icon)((Object[])obj)[1]);

        if(!isSelected) {
            setBackground(jlist.getBackground());
            setForeground(jlist.getForeground());
        }
        else {
            setBackground(jlist.getSelectionBackground());
            setForeground(jlist.getSelectionForeground());
        }

        return this;
    }
}
```

Eso es todo. Ahora el control de lista visualizará imágenes, como se muestra en la figura 13.13. Este ejemplo se encuentra en el archivo listimages.java del CD.

Procesamiento de doble clic en listas

Dice el programador novato: "Quiero permitir que el usuario lance un elemento desde un control de lista al hacer doble clic sobre él, pero el método valueChanged de la interfaz ListSelectionListener se llama para clics sencillos y no puedo decir si la lista ha recibido un clic doble". "Bien", decimos, "Puede utilizar los clics del ratón para interceptar los clics dobles". El programador novato dice, "¿De verdad? ¡Dígame más!"

Si interceptamos los eventos de ratón, podemos determinar el número de clics de ratón y, accediendo al modelo de la lista, podemos determinar cuál fue el elemento sobre el que se hizo clic.

Vamos a verlo en el código. Aquí, únicamente añadimos una lista con varios elementos a un applet y después añadimos un receptor de ratón a la lista:

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

/*
  <APPLET
   CODE=listdouble.class
   WIDTH=300
   HEIGHT=200 >
  </APPLET>
*/

public class listdouble extends JApplet implements MouseListener
{
    public void init()
    {
        Container contentPane = getContentPane();

          String[] items = {"Elemento 1", "Elemento 2", "Elemento 3",
"Elemento 4",
                            "Elemento 5", "Elemento 6", "Elemento 7",
"Elemento 8",
                           "Elemento 9", "Elemento 10", "Elemento 11",
"Elemento 12"};

        JList list = new JList(items);

        contentPane.setLayout(new FlowLayout());

        contentPane.add(new JScrollPane(list));

        list.addMouseListener(this);
    }

    ...
}
```

Ahora, en el método mouseClicked, determinamos dónde se hizo doble clic con el ratón; entonces utilizamos el método locationToIndex de la lista para determinar el elemento de la lista sobre el que se hizo clic. Finalmente, podemos determinar cuántas veces se hizo clic sobre el elemento con el método getClickCount de MouseEvent y, por tanto, informar del número de veces que se hizo clic sobre un elemento, como a continuación:

```
public void mouseClicked(MouseEvent e)
{
    JList jlist = (JList)e.getSource();
    int index = jlist.locationToIndex(e.getPoint());
```

```
            String outString = new String("Número de clics para el elemento
" +
            ++index + ": " + e.getClickCount());

            showStatus(outString);
        }

        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}
    }
```

El resultado se muestra en la figura 13.14. Como vemos, el applet informa del número de clics que se realizan sobre un elemento de lista cada vez. De esta forma, podemos manejar clics dobles e incluso triples. Este ejemplo se encuentra en el archivo listdouble.java del CD.

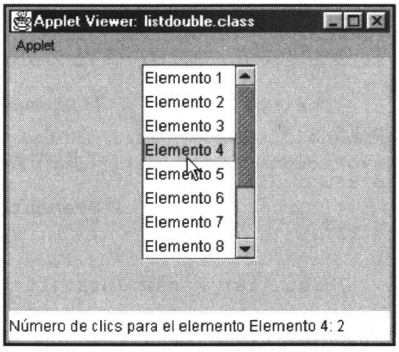


Figura 13.14. Gestión de doble clic en un control de lista.

14 Swing: barras, herramientas, cuadros, separadores y selectores

En este capítulo, veremos algunos controles importantes de la Swing: cuadros combinados, barras de progreso, separadores y selectores. También examinaremos la adición de herramientas de ayuda, unas pequeñas ventanas que muestran un texto de explicación cuando dejamos el ratón sobre un control. Todos estos temas, especialmente los cuadros combinados, son importantes en Java y los introduciremos antes de profundizar en el código.

Cuadros combinados

Los cuadros combinados son uno de los controles más importantes en la programación de la IU, pero el AWT no tiene un cuadro combinado. La Swing rectifica esto. Los cuadros combinados con combinaciones de campos de texto y listas desplazables. Son muy útiles, ya que permiten al usuario tanto seleccionar un elemento de la lista como introducir su propio valor en el campo de texto. Los cuadros combinados también son controles muy compactos, debido a que muestran únicamente un elemento y un botón que el

usuario puede utilizar para desplegar la lista de otros elementos. Los cuadros combinados pueden funcionar como listas desplegables, proporcionando al usuario un mecanismo incluso aún más compacto para visualizar los elementos de una lista que el que proporciona un cuadro de lista, y han pasado a ser muy populares por esa razón. De hecho, la Swing tiene como valor predeterminado del cuadro combinado que muestre únicamente una lista desplegable, ya que el valor predeterminado para los cuadros combinados sería hacerlos no editables, lo que los convierte en listas desplegables.

Barras de progreso

Las barras de progresos son controles relativamente nuevos, que se han hecho populares como mecanismo para proporcionar al usuario la indicación del progreso de una operación larga. Las barras de progreso, originalmente introducidas como mecanismo para mostrar el progreso de un programa de instalación, se utilizan ahora para todo tipo de operaciones de consumo de tiempo, como descargas de archivos desde la red Internet. Las barras de progreso muestran una barra coloreada dentro de ellas que crece (o se reduce), de forma similar a como lo hace el mercurio en un termómetro, para mostrar visualmente cómo progresa una operación. Puede orientar las barras de progreso tanto horizontal como verticalmente, seleccionar los colores y etiquetas que use en ellas y manejar sus eventos. Y aún más, las barras de progreso siguen siendo controles sencillos, ya que tienen únicamente una función: mostrar el progreso de una tarea. Examinaremos todas sus posibilidades en este capítulo.

Selectores

Como en la AWT, la Swing soporta cuadros de diálogo. Al contrario que la AWT, la Swing también soporta adicionalmente varios cuadros de diálogo que no tenemos que crear o personalizar nosotros mismos: selectores de archivos y selectores de color. Los selectores de archivos permiten al usuario seleccionar un archivo para abrir o guardar algo, muy similar a cualquier cuadro de diálogo de archivos estándar. Los selectores de color permiten al usuario seleccionar un color entre muchos. Ambos selectores representan cuadros de diálogo estándar y *Sun* únicamente nos ahorra tiempo al crearlos. Trabajaremos con ambos selectores en este capítulo. Podemos utilizar inmediatamente en los programas el selector de color, pero para utilizar los archi-

vos que devuelve el selector de archivos, tendremos que esperar unos pocos capítulos (hasta que comencemos a trabajar con archivos).

Herramientas de ayuda

Las herramientas de ayuda son esas ventanas pequeñas que aparecen y muestran un texto de explicación (como por ejemplo Descargar ahora o Abrir nueva carpeta) cuando se detiene el ratón sobre un control. Las herramientas de ayuda pueden ser muy útiles debido a que los usuarios de la IU tienen una gran resistencia a leer los manuales. Lo único que tiene que hacer el usuario es dejar el ratón sobre su programa para ver qué es lo que hacen los distintos controles. Por otro lado, tenga en cuenta que muchas herramientas de ayuda (conectadas con muchos elementos de texto en un control de texto, por ejemplo) pueden ser contraproducentes y dar una apariencia de dificultad a su programa.

Separadores

Los separadores son barras horizontales o verticales que le permiten organizar sus controles en grupos. Aunque se utilizan mayoritariamente en los menús para dividir elementos de menú en agrupaciones lógicas, podrá utilizar separadores en componentes JApplet y JFrame como cualquier otro control, como veremos más adelante.

Este es el resumen de lo que aparece en este capítulo. Como puede ver, vendrá mucho más. Es el momento de pasar al primer punto que comienza con los cuadros combinados.

Creación de cuadros combinados

"Mmh", dice el programador novato, "quiero permitir al usuario seleccionar una palabra para realizar un análisis sintáctico a partir de una lista de palabras, pero ahora los usuarios dicen que quieren introducir sus propias palabras también". "Suena razonable, ¿qué tal si utiliza un cuadro combinado?" "Bien", dice el PN, "¿puede añadir uno a mi código?".

Los cuadros combinados se pueden utilizar en la Swing de dos formas: como cuadros combinados normales y como listas desplegables. Las listas desplegables permiten al usuario seleccionar en una lista que aparece cuando

hace clic sobre la flecha que apunta hacia abajo. Los cuadros combinados normales, por otra parte, son combinaciones de campos de texto y listas desplegables; los usuarios pueden seleccionar un elemento de la lista desplegable o introducir su propio texto en el campo de texto. Observe que al contrario de los cuadros de lista, únicamente se puede seleccionar un elemento a la vez en un cuadro combinado.

Los cuadros combinados están soportados por la clase JComboBox en la Swing y aquí tenemos el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JComboBox
```

La tabla 14.1 muestra los campos de la clase JComboBox, la tabla 14.2 muestra sus constructores y la tabla 14.3 sus métodos.

Los datos actuales del cuadro combinado se almacenan en su modelo, que es un objeto de la clase DefaultComboBoxModel predeterminado. Puede encontrar los métodos de este objeto en la tabla 14.4. Observe que para añadir o eliminar elementos, puede utilizar los métodos del modelo, como se muestra en la tabla. Para obtener el objeto del modelo, puede utilizar el método getModel del JComboBox.

Tabla 14.1. Campos de la clase JComboBox.

Campo	Descripción
protected String actionCommand	El comando de acción.
protected ComboBoxModel dataModel	El modelo de datos.
protected ComboBoxEditor editor	El editor, que es responsable del campo de texto.
protected boolean isEditable	Indica si el cuadro combinado es editable.
protected JComboBox.KeySelectionManager keySelectionManager	El gestor de selección de teclas.
protected boolean lightWeightPopupEnabled	Indica si la lista desplegable está habilitada como componente de poco peso.

Campo	Descripción
protected int maximumRowCount	Contiene la cuenta de filas.
protected ListCellRenderer renderer	El renderizador de celdas.
protected Object selectedItem Reminder	Un recuerdo del elemento seleccionado.

Tabla 14.2. Constructores de la clase JComboBox.

Constructor	Descripción
JComboBox()	Construye un objeto JComboBox.
JComboBox(ComboBoxModel aModel)	Construye un objeto JComboBox que toma elementos de un modelo de cuadro combinado ya existente.
JComboBox(Object[] items)	Construye un objeto JComboBox que contiene los elementos en la matriz indicada.
JComboBox(Vector items)	Construye un objeto JComboBox que contiene los elementos en el vector indicado.

Tabla 14.3. Métodos de la clase JComboBox.

Método	Descripción
void actionPerformed (ActionEvent e)	Público como implementación de un efecto lateral.
void addActionListener (ActionListener l)	Añade un receptor de acciones.
void addItem(Object anObject)	Añade el elemento a la lista de elementos.
void addItemListener(ItemListener aListener)	Añade un receptor de elementos.
void configureEditor(ComboBox Editor anEditor, Object anItem)	Inicializa el editor.
void contentsChanged (ListDataEvent e)	Público como implementación de un efecto lateral.

Método	Descripción
protected JComboBox.Key SelectionManager createDefault KeySelectionManager()	Obtiene en instancia del gestor de selección de teclas predeterminado.
protected void fireActionEvent()	Notifica a todos los receptores que han registrado interés en la notificación de este tipo de evento.
protected void fireItemState Changed(ItemEvent e)	Notifica a todos los receptores que han registrado interés para la notificación en este tipo de evento.
AccessibleContext getAccessible Context()	Obtiene el contexto accesible.
String getActionCommand()	Obtiene el comando de acción.
ComboBoxEditor getEditor()	Obtiene el editor utilizado para editar el elemento seleccionado del campo de texto JComboBox.
Object getItemAt(int index)	Obtiene la lista de elementos en el índice indicado.
int getItemCount()	Obtiene los elementos de la lista.
JComboBox.KeySelectionManager getKeySelectionManager()	Obtiene el gestor de selección de teclas de la lista.
int getMaximumRowCount()	Obtiene el máximo número de elementos que puede visualizar a la vez un cuadro combinado.
ComboBoxModel getModel()	Obtiene el modelo de datos.
ListCellRenderer getRenderer()	Obtiene el renderizador.
int getSelectedIndex()	Obtiene el índice del elemento actualmente seleccionado.
Object getSelectedItem()	Obtiene el elemento seleccionado actual.
Object[] getSelectedObjects()	Obtiene una matriz que contiene los elementos seleccionados.
ComboBoxUI getUI()	Obtiene el objeto apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza éste componente.

Método	Descripción
void hidePopup()	Provoca que el cuadro combinado cierre su ventana emergente.
void insertItemAt(Object anObject, int index)	Inserta elementos en la lista de elementos en un índice dado.
protected void installAncestor Listener()	Instala un receptor padre.
void intervalAdded (ListDataEvent e)	Invocado cuando los elementos se añaden al modelo de datos interno.
void intervalRemoved (ListDataEvent e)	Invocado cuando los valores se han eliminado de un modelo de datos.
boolean isEditable()	Devuelve cierto si el objeto JComboBox es editable.
boolean isFocusTraversable()	Devuelve cierto si el componente puede recibir el foco.
boolean isLightWeight-PopupEnabled()	Devuelve cierto si los cuadros emergentes de poco peso se utilizan. Devuelve falso si se utilizan diálogos emergentes de más peso.
boolean isPopupVisible()	Determina la visibilidad del cuadro emergente.
protected String paramString()	Obtiene una representación de cadena para este objeto JComboBox.
void processKeyEvent(KeyEvent e)	Procesa a eventos de teclas, buscando la tecla Tab.
void removeActionListener (ActionListener l)	Elimina un receptor de acción.
void removeAllItems()	Elimina todos los elementos de la lista de elementos.
void removeItem(Object anObject)	Elimina un elemento de la lista de elementos.
void removeItemAt(int anIndex)	Elimina elemento en el index. Este método funciona únicamente si el objeto JComboBox utiliza el modelo de datos predeterminado.
void removeItemListener (ItemListener aListener)	Elimina al receptor de elementos.

Método	Descripción
protected void selectedItemChanged()	Llamado cuando cambia el elemento seleccionado.
boolean selectWithKeyChar(char keyChar)	Selecciona el elemento de la lista que corresponde al carácter de teclado indicado.
void setActionCommand(String aCommand)	Asigna el comando de acción que debería incluirse en el evento enviado a los receptores de acción.
void setEditable(boolean aFlag)	Asigna el indicador de edición.

Tabla 14.4. Métodos de la clase DefaultComboBoxModel.

Método	Descripción
void addElement(Object anObject)	Añade un elemento al final del modelo.
Object getElementAt(int index)	Obtiene el valor en el índice indicado.
int getIndexOf(Object anObject)	Obtiene la posición del índice del objeto indicado en la lista.
Object getSelectedItem()	Devuelve el elemento seleccionado.
int getSize()	Obtiene la longitud de la lista.
void insertElementAt(Object anObject, int index)	Añade un elemento al final del modelo.
void removeElementAt(int index)	Elimina un elemento en un índice específico.
void setSelectedItem(Object anObject)	Asigna el elemento seleccionado.

Para añadir elementos a un cuadro combinado, puede utilizar el conveniente método addItem, pasándole un objeto que a su vez pasará a su modelo interno. Veamos un ejemplo que muestra cómo funciona. En este caso, basta crear un cuadro combinado y cinco elementos (es decir, cinco objetos String). Aquí tenemos el código:

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

/*
  <APPLET
    CODE = combobox.class
```

```
    WIDTH = 200
    HEIGHT = 200 >
  </APPLET>
*/

public class combobox extends JApplet
{
    private JComboBox jcombobox = new JComboBox();

    public void init()
    {
        Container contentPane = getContentPane();

        jcombobox.addItem("Elemento 1");
        jcombobox.addItem("Elemento 2");
        jcombobox.addItem("Elemento 3");
        jcombobox.addItem("Elemento 4");
        jcombobox.addItem("Elemento 5");

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jcombobox);
    }
}
```

El resultado se muestra en la figura 14.1, donde puede ver el cuadro combinado con su lista desplegable abierta, mostrando los cinco elementos. Cuando el usuario selecciona un elemento de esa lista y suelta el botón del ratón, el elemento queda como nuevo elemento seleccionado en el cuadro combinado y también aparece en el campo de texto. Este ejemplo se encuentra en el archivo combobox.java del CD que acompaña a este libro. Observe que, por defecto, el campo de texto en el cuadro combinado no es editable; mostraremos cómo cambiarlo en unas pocas páginas.

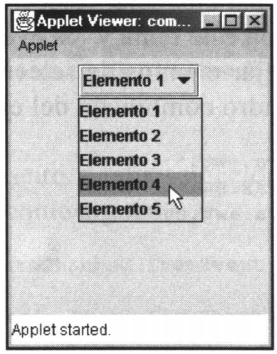


Figura 14.1. Creación de un cuadro combinado.

Por tanto, ¿cómo recuperar elementos de un cuadro combinado? Debido a que únicamente se puede seleccionar al mismo tiempo un elemento de un

cuadro combinado, puede obtener ese elemento con el método getSelectedItem, que devuelve el mismo elemento (observe que el elemento es un objeto). También puede utilizar getSelectedIndex para determinar el índice del elemento seleccionado en curso en el cuadro en la lista del cuadro combinado, como sigue:

```
String outString = (String)jcombobox.getSelectedItem()
int index = (String) jcombobox.getSelectedIndex()
```

Para obtener un elemento en un índice específico, utilice el método getElementAt del modelo. También puede utilizar los métodos insertElementAt y removeElementAt del modelo para editar la lista.

Observe que también puede responder a los eventos del cuadro combinado. Lo examinaremos en el siguiente tema.

Manejo de los eventos de selección del cuadro combinado

El programador novato vuelve y dice, "Quiero cambiar el color de dibujo en mi nuevo programa tan pronto como el usuario seleccione un nuevo color de un cuadro combinado; ¿cómo puedo hacerlo?". "Es fácil", respondemos, "basta con hacer que el código responda a eventos de selección". "Ajá", dice el PN. "Aquí tiene mi código, ¿puede arreglarlo?".

Puede utilizar dos tipos de receptores con cuadros combinados: receptores de acción para manejar eventos de edición en el cuadro de texto y receptores de elementos para manejar selecciones de listas. Examinaremos las selecciones de listas en este tema y los eventos de edición en el siguiente.

Para manejar eventos de selección, basta con añadir un receptor de elementos al cuadro combinado del ejemplo desarrollado en el tema previo:

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

/*
  <APPLET
    CODE = comboboxevents.class
    WIDTH = 300
    HEIGHT = 200 >
  </APPLET>
*/

public class comboboxevents extends JApplet implements ItemListener
```

```
{
    JComboBox jcombobox = new JComboBox();
    String outString = "";

    public void init()
    {
        Container contentPane = getContentPane();

        jcombobox.addItem("Elemento 1");
        jcombobox.addItem("Elemento 2");
        jcombobox.addItem("Elemento 3");
        jcombobox.addItem("Elemento 4");
        jcombobox.addItem("Elemento 5");

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jcombobox);

        jcombobox.addItemListener(this);
    }
  ...
}
```

Ahora añadimos el método itemStateChanged a este applet que necesita la interfaz ItemListener:

```
public void itemStateChanged(ItemEvent e)
{
...
}
```

Este método recibe un objeto de la clase ItemEvent y puede determinar si el elemento correspondiente en la lista del cuadro combinado estaba seleccionando, utilizando el método getStateChange como sigue:

```
public void itemStateChanged(ItemEvent e)
{
    if(e.getStateChange() == ItemEvent.SELECTED)
...
}
```

Indicaremos si un elemento estaba seleccionado o deseleccionado en la barra de estado del applet como sigue:

```
public void itemStateChanged(ItemEvent e)
{
    if(e.getStateChange() == ItemEvent.SELECTED)
        outString += "Seleccionado: " + (String)e.getItem();
    else
        outString += "Deseleccionado: " + (String)e.getItem();

    showStatus(outString);
}
```

El resultado de este código se muestra en la figura 14.2. Como puede ver en el pie del applet de la figura, el código informa cada vez que el usuario deselecciona un elemento y selecciona otro. Este programa es un éxito y se encuentra en el archivo comboboxevents.java del CD.

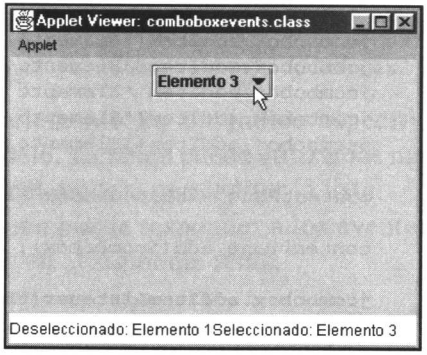


Figura 14.2. Trabajo con eventos de selección del cuadro combinado.

Creación de cuadros combinados editables

"Maldición", dice el programador novato, "no puedo hacer que el campo de texto de mi cuadro combinado funcione; parece que no puedo introducir ningún texto en él". Sonreímos y contestamos, "Eso es debido a que lo tiene como *editable*". "Ah", dice el PN. "¿Cómo puedo hacerlo?".

Para convertir el campo de texto en un cuadro combinado editable, utilice el método setEditable. Para manejar eventos de edición, utilice un receptor de acciones.

Aquí tenemos un ejemplo donde hacemos el cuadro combinado editable y le añadimos un receptor de acción:

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

/*
  <APPLET
    CODE = comboboxedit.class
    WIDTH = 200
    HEIGHT = 200 >
  </APPLET>
*/

public class comboboxedit extends JApplet implements ActionListener
{
```

```
    private JComboBox jcombobox = new JComboBox();

    public void init()
    {
        Container contentPane = getContentPane();

        jcombobox.addItem("Elemento 1");
        jcombobox.addItem("Elemento 2");
        jcombobox.addItem("Elemento 3");
        jcombobox.addItem("Elemento 4");
        jcombobox.addItem("Elemento 5");

        jcombobox.setEditable(true);

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jcombobox);

        jcombobox.getEditor().addActionListener(this);

    }
  ...
}
```

Observe que no estamos añadiendo un receptor de acción al cuadro combinado directamente sino, más bien, al *editor* del cuadro combinado, que es un objeto que implementa la interfaz ComboBoxEditor. Obtenemos este objeto con el método getEditor. El editor es responsable de las acciones de edición en el campo de texto del cuadro combinado, y la tabla 4.5 muestra los métodos de ComboBoxEditor.

En este ejemplo, mostramos el texto del elemento antes de que se haya editado y el texto después de que haya sido editado y el usuario pulse Intro. Para obtener el texto del elemento antes de su edición, puede utilizar el método getSelectedItem y para obtener el elemento después de su edición, puede utilizar el método getItem, como sigue:

```
    public void actionPerformed(ActionEvent e)
    {
        String outString = (String)jcombobox.getSelectedItem()
          + " se cambió a " + (String)jcombobox.getEditor().getItem();

        showStatus(outString);
    }
```

La figura 14.3 muestra el resultado. Cuando el usuario edita un elemento en el campo de texto y pulsa Intro, el texto viejo y nuevo del cuadro combinado aparecen en la barra de estado del applet, como muestra la figura. Este ejemplo se encuentra en el archivo comboboxedit.java del CD.

Tabla 14.5. Métodos de la interfaz ComboBoxEditor.

Método	Descripción
void addActionListener(ActionListener l)	Añade un receptor de acciones.
Component getEditorComponent()	Devuelve el componente del editor.
Object getItem()	Devuelve el elemento editado.
void removeActionListener(ActionListener l)	Elimina al receptor de acciones.
void selectAll()	Solicita al editor el inicio de la edición y selecciona todo.
void setItem(Object anObject)	Asigna el elemento que debería ser editado.

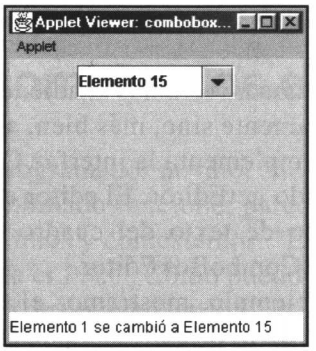


Figura 14.3. Edición del campo de texto de un cuadro combinado.

Adición de imágenes a cuadros combinados

"Veamos", dice el programador novato, "¿existe alguna posibilidad de mostrar imágenes en un cuadro combinado?" "Claro", decimos, "basta con sentarse y examinarlo".

Puede añadir imágenes a cuadros combinados al igual que puede hacerlo con cuadros de lista. De hecho, cuando añade imágenes a los cuadros combinados, realmente añade imágenes a la parte de lista del cuadro combinado, por lo que el proceso es prácticamente igual que añadir imágenes a las listas.

Aquí tenemos un ejemplo en el que creamos un nuevo modelo, newModel, y el renderizador de celdas, newRenderer, para un cuadro combinado. Aquí vemos cómo instalarlo en el cuadro combinado:

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

/*
  <APPLET
    CODE = comboimages.class
    WIDTH = 400
    HEIGHT = 200 >
  </APPLET>
*/

public class comboimages extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        newModel newmodel = new newModel();
        newRenderer newrenderer = new newRenderer();

        JComboBox jlist = new JComboBox(newmodel);
        jlist.setRenderer(newrenderer);

        contentPane.add(new JScrollPane(jlist));
    }
}
```

Este nuevo modelo y renderizador almacenará y visualizará las imágenes en un cuadro combinado y la figura 14.4 muestra el resultado. Ahora, lo único que resta es crear el nuevo modelo y el nuevo renderizador y lo haremos en los dos temas siguientes. Este ejemplo se encuentra en el archivo comboimages.java del CD.

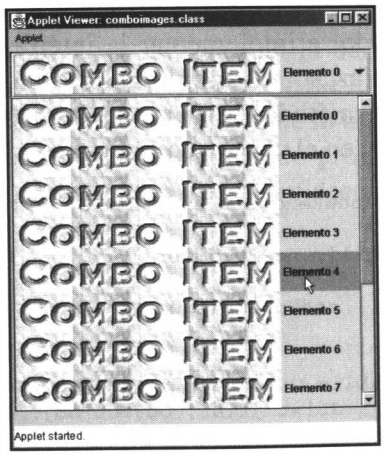


Figura 14.4. Adición de imágenes a cuadros combinados.

Creación de un modelo de cuadro combinado

Crearemos un nuevo modelo de cuadro combinado en este tema para utilizarlo con el cuadro combinado del tema anterior (el que visualiza imágenes). Este nuevo modelo, que extiende la clase DefaultComboBoxModel, utilizará el método addElement para almacenar los elementos de datos en el modelo.

Cada elemento se compone de una cadena de texto y un icono de imagen como sigue:

```
class newModel extends DefaultComboBoxModel
{
    public newModel()
    {
        for(int loop_index = 0; loop_index <= 12; loop_index++) {
            addElement(new Object[] {"Elemento " + loop_index,
                new ImageIcon("combo.jpg")});
        }
    }
}
```

Creación de un renderizador personalizado para el cuadro combinado

En este tema, crearemos un renderizador que visualice imágenes en un cuadro combinado (este renderizador se utiliza en el código de los dos temas anteriores). Para crear el renderizador de celdas para un cuadro combinado, basta extender la clase JLabel, que es la clase de cada elemento en un cuadro combinado, e implementar la interfaz ListCellRenderer. Para implementar esa interfaz, sobrescribimos el método getListCellRendererComponent utilizando los métodos setText y setIcon de la clase JLabel para instalar el texto e imágenes en el cuadro combinado:

```
class newRenderer extends JLabel implements ListCellRenderer
{
    public newRenderer()
    {
        setOpaque(true);
    }

    public Component getListCellRendererComponent(
        JList jlist, Object obj, int index, boolean isSelected,
        boolean focus)
    {
```

```
        newModel model = (newModel)jlist.getModel();

        setText((String)((Object[])obj)[0]);
        setIcon((Icon)((Object[])obj)[1]);

        if(!isSelected) {
            setBackground(jlist.getBackground());
            setForeground(jlist.getForeground());
        }
        else {
            setBackground(jlist.getSelectionBackground());
            setForeground(jlist.getSelectionForeground());
        }

        return this;
    }
}
```

El resultado se muestra en la figura 14.4, donde puede ver las imágenes visualizadas en un cuadro combinado.

Creación de barras de progreso

"Bien", dice el programador novato, "es cierto que lleva tres horas el que mi programa ordene sus datos internos, pero ¿existe alguna razón para que los usuarios lo comprendan?" "Por supuesto", decimos, "a menos que le dé algún tipo de indicación visual de lo que sucede. ¿Qué tal si añadimos una barra de progreso a su programa?". "¡Claro! ", dice el PN. "¿Qué es una barra de progreso?"

Un control de barra de progreso dibuja una barra coloreada y actualizada dentro de sí mismo para visualizar el progreso de alguna operación. Puede configurar los valores mínimo y máximo de la barra de progreso cuando llama al constructor JProgressBar, o puede utilizar los métodos setMinimum y setMaximum. Puede actualizar repetidamente la barra de progreso llamando a su método setValue para indicar el estado progresivo de la operación. Aquí tenemos el diagrama de herencia para la clase JProgressBar:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JProgressBar
```

Puede encontrar los campos de la clase JProgressBar mostrados en la tabla 14.6 (observe que también hereda dos constantes de la clase SwingConstants que utilizamos para la orientación: HORIZONTAL y VERTICAL), sus constructores se muestran en la tabla 14.7 y sus métodos se muestran en la tabla 14.8.

Tabla 14.6. Campos de la clase JProgressBar.

Campo	Descripción
protected ChangeEvent changeEvent	El evento de cambio.
protected ChangeListener changeListener	El receptor de cambio.
protected BoundedRangeModel model	El modelo de datos que contiene los valores de datos para la barra de progreso.
protected int orientation	La orientación en que se visualiza la barra de progreso.
protected boolean paintBorder	Devuelve cierto si debe existir un borde alrededor de la barra de progreso.
protected boolean paintString	Devuelve cierto si debe existir una etiqueta de cadena en la barra de progreso.
protected String progressString	Cadena opcional que puede mostrarse en la barra de progreso.

Tabla 14.7. Constructores de la clase JProgressBar.

Constructor	Descripción
JProgressBar()	Construye una barra de progreso horizontal.
JProgressBar(BoundedRangeModel newModel)	Construye una barra de progreso horizontal (orientación predeterminada).
JProgressBar(int orient)	Construye una barra de progreso con la orientación indicada, que puede ser o bien JProgressBar.VERTICAL o JProgressBar.HORIZONTAL.
JProgressBar(int min, int max)	Construye una barra de progreso horizontal, que es la predeterminada.

Constructor	Descripción
JProgressBar(int orient, int min, int max)	Construye una barra de progreso utilizando la orientación y los valores mínimo y máximo indicados.

Tabla 14.8. Métodos de la clase JProgressBar.

Método	Descripción
void addChangeListener(ChangeListener l)	Añade un receptor de cambios.
protected ChangeListener createChangeListener()	Crea un receptor de cambios.
protected void fireStateChanged()	Notifica a todos los receptores que registraron interés para la notificación de este tipo de evento.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible asociado con este objeto JComponent.
int getMaximum()	Obtiene el valor máximo del modelo.
int getMinimum()	Obtiene el valor mínimo del modelo.
BoundedRangeModel getModel()	Obtiene el modelo de datos.
int getOrientation()	Devuelve JProgressBar.VERTICAL o JProgressBar.HORIZONTAL.
double getPercentComplete()	Obtiene el porcentaje completado para la barra de progreso.
String getString()	Obtiene el valor en curso de la cadena de progreso.
ProgressBarUI getUI()	Obtiene el objeto apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza este componente.
int getValue()	Obtiene el valor actual del modelo.
boolean isBorderPainted()	Devuelve cierto si la barra de progreso tiene un borde y falso en otro caso.
boolean isStringPainted()	Devuelve cierto si la cadena se va a dibujar en la barra de progreso.

Método	Descripción
protected void paintBorder(Graphics g)	Pinta el borde de la barra de progreso (siempre que la propiedad BorderPainted sea cierto).
protected String paramString()	Obtiene una representación de cadena del objeto JProgressBar.
void removeChangeListener(ChangeListener l)	Elimina un receptor de cambios del botón.
void setBorderPainted(boolean b)	Asigna si la barra de progreso debería pintar su borde.
void setMaximum(int n)	Asigna el valor máximo del modelo a n.
void setMinimum(int n)	Asigna el valor mínimo del modelo a n.
void setModel(BoundedRangeModel newModel)	Asigna al modelo de datos utilizado por el objeto JProgressBar.
void setOrientation(int newOrientation)	Asigna la orientación de la barra de progreso a newOrientation.
void setString(String s)	Asigna el valor de la cadena de progreso.
void setStringPainted(boolean b)	Asigna si la barra de progreso renderiza una cadena.
void setUI(ProgressBarUI ui)	Asigna el objeto apariencia que renderiza este componente.
void setValue(int n)	Asigna el valor actual del modelo a n.
void updateUI()	Llamada por la clase UIFactory para indicar que la apariencia ha cambiado.

Existen diversas formas de visualizar las barras de progreso. Por ejemplo, puede hacerlas horizontales o verticales cuando llama al constructor de la clase o utilizar el método setOrientation. También puede seleccionar el color de dibujo para la barra actual, visualizar la etiqueta dentro de la barra de progreso que indique el valor en curso, y más. El uso básico de todas las barras de progreso, no obstante, es el mismo sin importar la apariencia que tengan; asignamos un valor mínimo y máximo a la barra de progreso (los valores predeterminados son 0 y 100) y actualizamos su pantalla con el método setValue según convenga. Aquí tenemos unos pocos ejemplos que muestran algunas de las formas en que podemos mostrar las barras de progreso:

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

/*
  <APPLET
    CODE = progressbar.class
    WIDTH = 500
    HEIGHT = 200 >
  </APPLET>
*/

public class progressbar extends JApplet
{
    JProgressBar jprogressbar1, jprogressbar2, jprogressbar3,
        jprogressbar4, jprogressbar5, jprogressbar6;

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        jprogressbar1 = new JProgressBar();
        jprogressbar1.setValue(50);
        contentPane.add(jprogressbar1);

        jprogressbar2 = new JProgressBar();
        jprogressbar2.setMinimum(100);
        jprogressbar2.setMaximum(200);
        jprogressbar2.setValue(180);
        jprogressbar2.setForeground(Color.red);
        contentPane.add(jprogressbar2);

        jprogressbar3 = new JProgressBar();
        jprogressbar3.setOrientation(JProgressBar.VERTICAL);
        jprogressbar3.setForeground(Color.blue);
        jprogressbar3.setValue(50);
        jprogressbar3.setStringPainted(true);
    jprogressbar3.setBorder(BorderFactory.createRaisedBevelBorder());
        contentPane.add(jprogressbar3);

        jprogressbar4 = new JProgressBar();
        jprogressbar4.setOrientation(JProgressBar.VERTICAL);
        jprogressbar4.setForeground(Color.red);
        jprogressbar4.setValue(80);
        jprogressbar4.setStringPainted(true);
        jprogressbar4.setBorderPainted(false);
        contentPane.add(jprogressbar4);

        jprogressbar5 = new JProgressBar();
        jprogressbar5.setOrientation(JProgressBar.VERTICAL);
        jprogressbar5.setStringPainted(true);
        jprogressbar5.setString("¡Hola desde Swing!");
        jprogressbar5.setValue(90);
        contentPane.add(jprogressbar5);
```

```
        }
    }
```

El resultado de este código se muestra en la figura 14.5, donde puede ver varias barras de progreso visualizadas de formas distintas. Aquí tenemos muchas posibilidades (observe, en concreto, que puede añadir una etiqueta a la barra de progreso para indicar su configuración actual y puede personalizar el borde). Este ejemplo se encuentra el en archivo progressbar.java del CD.

Por otro lado, las barras de progreso estáticas, como las que muestra la figura 14.5, no tienen mucha utilidad ya que no hacen nada. Para ver cómo actualizar una barra de progreso, examine el tema siguiente.

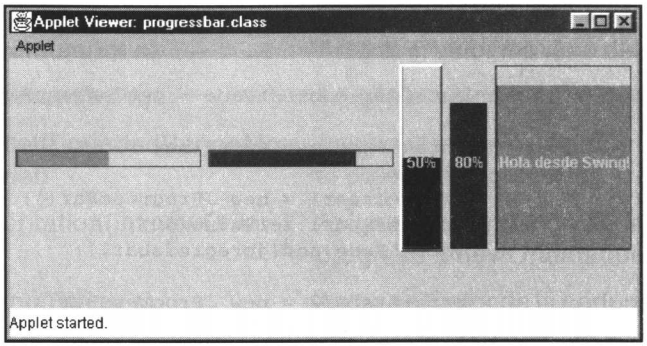


Figura 14.5. Visualización de barras de progreso.

Actualización de barras de progreso

El programador novato dice, "Vaya, he puesto barras de progreso en mi programa, pero no hacen nada; ¿qué está mal?" "Bien", respondemos, "eso se debe a que tiene que llamar al método setValue para actualizarlas". "Ajá", dice el PN.

Después de la creación y visualización de una barra de color de progreso, es cuenta del programador actualizarla con el método setValue. Normalmente lo haremos dividiendo la tarea que consuma tiempo en partes y actualizaremos la barra de progreso a medida que se complete cada parte.

Aquí tenemos un ejemplo (progressbarupdate.java en el CD) que incrementa una barra de progreso cada vez que se pulsa un botón:

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

/*
```

```
  <APPLET
    CODE = progressbarupdate.class
    WIDTH = 400
    HEIGHT = 200 >
  </APPLET>
*/

public class progressbarupdate extends JApplet
{
    JProgressBar jprogressbar = new JProgressBar();
    JButton jbutton = new JButton("Incrementar la barra de progreso");

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jprogressbar);
        contentPane.add(jbutton);

        jprogressbar.setStringPainted(true);

        jbutton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                jprogressbar.setValue(jprogressbar.getValue() + 10);
            }
        });
    }
}
```

El resultado se muestran en la figura 14.6. Observe que puede actualizar la barra de progreso solamente haciendo clic sobre el botón. Normalmente, por supuesto, su código actualizará la barra de progreso automáticamente, debido a que las barras de progreso se utilizan para indicar al usuario el progreso de las operaciones que consumen tiempo.

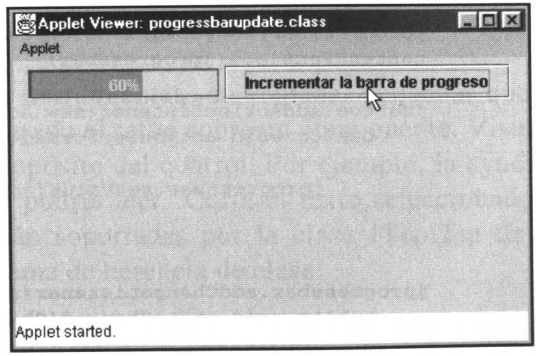


Figura 14.6. Actualización de una barra de progreso.

Manejo de los eventos de barras de progreso

Los eventos de barras de progreso ocurren cada vez que el valor de la barra de progreso cambia, y podemos trabajar con receptores de cambios para capturar esos eventos. Aquí tenemos un ejemplo (progressbarevents.java en el CD) que permite al usuario actualizar la barra de progreso con un botón. Captura los eventos a medida que suceden. Informamos del nuevo valor de la barra de progreso cada vez que cambia, junto con sus valores mínimo y máximo, como se muestra a continuación:

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

/*
  <APPLET
    CODE = progressbarevents.class
    WIDTH = 400
    HEIGHT = 200 >
  </APPLET>
*/

public class progressbarevents extends JApplet
{
    JProgressBar jprogressbar = new JProgressBar();
    JButton jbutton = new JButton("Incrementar la barra de progreso");

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jbutton);

        jprogressbar.setForeground(Color.blue);
        contentPane.add(jprogressbar);

        jbutton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                jprogressbar.setValue(jprogressbar.getValue() + 10);
            }
        });

        jprogressbar.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e)
            {
                    showStatus("Mínimo de la barra de progreso: " +
jprogressbar.getMinimum() +
```

```
                    " máximo: " + jprogressbar.getMaximum() +
                    " valor: " + jprogressbar.getValue());
                }
            });
        }
    }
```

El resultado de éste código se muestra la figura 14.7. Como vemos, cada vez que el usuario actualiza la barra de progreso, su nuevo valor aparece en la barra de estado del applet. Este ejemplo es un éxito y puede encontrarlo en el archivo progressbarevents.java del CD.

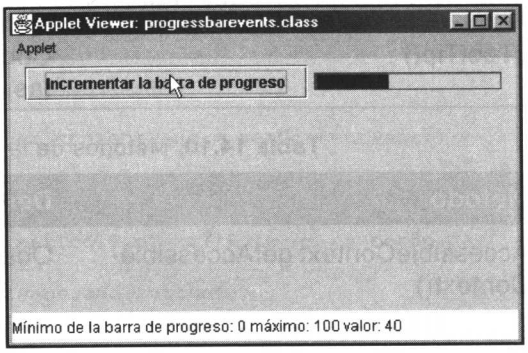


Figura 14.7. Manejo de eventos de barras de progreso.

Creación de ayudas de herramientas

"Bien", dice el especialista de soporte de productos, "no parece que nadie lea nunca los manuales. ¿Existe alguna forma de hacer que nuestros programas sean autoexplicativos?" "Bien", decimos, "podría reducir todo a una serie de animaciones". "Mm", dice el ESP pensativo. "Es una broma", contestamos rápidamente. "Añadiremos ayudas de herramientas a los controles del programa".

Las ayudas de herramientas son ventanas pequeñas que aparecen cuando el usuario deja parado el ratón sobre un componente. Visualizan algún texto que explica el propósito del control. Por ejemplo, la ayuda de herramientas del botón Cortar podría leer "Corta el texto seleccionado". Las ayudas de herramientas están soportadas por la clase JToolTip de la Swing y aquí tenemos el diagrama de herencia de clase:

```
java.lang.Object
  |
  +--java.awt.Component
```

```
|
+-java.awt.Container
      |
      +-javax.swing.JComponent
            |
            +-javax.swing.JToolTip
```

La tabla 14.9 muestra el constructor de la clase JToolTip y la tabla 14.10 muestra sus métodos.

Tabla 14.9. El constructor de la clase JToolTip.

Constructor	Descripción
JToolTip()	Construye una ayuda de herramientas.

Tabla 14.10. Métodos de la clase JToolTip.

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
JComponent getComponent()	Obtiene el elemento al que se aplica la ayuda de herramientas.
String getTipText()	Obtiene el texto que se muestra en la ayuda de herramientas.
ToolTipUI getUI()	Obtiene el objeto apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza este componente.
protected String paramString()	Obtiene una representación de cadena de este objeto JToolTip.
void setComponent(JComponent c)	Asigna el componente que describe esta ayuda de herramientas.
void setTipText(String tipText)	Asigna al texto mostrado cuando aparece la ayuda de herramientas.
void updateUI()	Llamada por la clase UIFactory cuando cambia la apariencia.

Muchos componentes ya tienen un método setToolTipText que puede utilizar para añadir una ayuda de herramientas. Aquí tenemos un ejemplo en el que añadimos una ayuda de herramientas a un botón:

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

/*
  <APPLET
    CODE=tooltip.class
    WIDTH=300
    HEIGHT=200 >
  </APPLET>
*/

public class tooltip extends JApplet {
    JButton button = new JButton("Haz clic aquí");
    JTextField text = new JTextField(20);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        button.setToolTipText("Esto es un botón.");

        contentPane.add(button);
        contentPane.add(text);

        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event) {
                text.setText("¡Hola desde Swing!");
            }
        });
    }
}
```

Es tan sencillo como añadir una ayuda de herramientas a la mayoría de los componentes. El resultado se muestra en la figura 14.8, donde puede ver la ayuda de herramientas con su texto explicativo. Este ejemplo se encuentra en el archivo tooltip.java del CD.

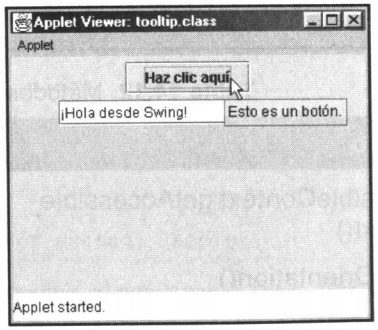


Figura 14.8. Uso de ayudas de herramientas.

Creación de separadores

El programador novato dice, "El gran jefe me ha dicho que hay demasiados campos de texto en mi programa y debería dividirlos en grupos". "Puede hacerlo de diversas formas sencillas", respondemos, "como añadir campos de texto a los paneles y dar a los paneles varios colores de fondo. Puede incluso utilizar separadores. Para empezar, ¿Cuántos campos de texto existen en su programa?" "Cerca de 2.413", dice el PN. "Ojo", decimos.

Los separadores son líneas verticales u horizontales y normalmente aparecen en los menús para separar elementos de menú en grupos lógicos, pero también pueden utilizarse para separar componentes en una distribución. Quedan soportados por la clase JSeparator en la Swing, y aquí tenemos el diagrama de herencia de clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JSeparator
```

La tabla 14.11 muestra los constructores de la clase JSeparator y la tabla 14.12 muestra sus métodos.

Tabla 14.11. Constructores de la clase JSeparator.

Constructor	Descripción
JSeparator()	Crea un nuevo separador horizontal.
JSeparator(int orientation)	Crea un nuevo separador con la orientación horizontal o vertical indicada.

Tabla 14.12. Métodos de la clase JSeparator.

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el contexto accesible.
int getOrientation()	Obtiene la orientación de este separador.

Método	Descripción
SeparatorUI getUI()	Obtiene el objeto apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza este componente.
boolean isFocusTraversable()	Indica si este componente puede recibir el foco.
protected String paramString()	Obtiene una representación de cadena de este objeto JSeparator.
void setOrientation(int orientation)	Asigna la orientación del separador.
void setUI(SeparatorUI ui)	Asigna el objeto apariencia que renderiza este componente.
void updateUI()	Llamada por la clase UIFactory cuando cambia la apariencia.

Vamos a examinar un ejemplo. Aquí, situamos un separador entre los campos de texto en una distribución de flujo. Para hacer visible el separador, tenemos que hacer algo más que añadirlo a la distribución; también debemos proporcionar un tamaño preferente utilizando el método setPreferredSize. La forma normal de hacerlo es utilizar el método getPreferredSize del separador para obtener el ancho actual del separador y después pasarlo con el nuevo tamaño que queramos en el método setPreferredSize. Los métodos setPreferredSize y setPreferredSize trabajan con objetos de la clase Dimension de la AWT, que tiene los campos: width y height. Aquí tenemos la forma de crear un nuevo separador y obtener sus dimensiones:

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

/*
  <APPLET
    CODE = separator.class
    WIDTH = 400
    HEIGHT = 200 >
  </APPLET>
*/

public class separator extends JApplet
{
    JSeparator jseparator = new JSeparator(JSeparator.VERTICAL);
    Dimension dimension = jseparator.getPreferredSize();
```

```
    ...
}
```

Ahora podemos poner el separador entre los campos de texto, como hacemos aquí, dándole una altura de cien puntos:

```
public class separator extends JApplet
{
    JSeparator jseparator = new JSeparator(JSeparator.VERTICAL);
    Dimension dimension = jseparator.getPreferredSize();

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(new JTextField("¡Hola desde Swing!"));
        contentPane.add(jseparator);
        contentPane.add(new JTextField("¡Hola desde Swing!"));

     jseparator.setPreferredSize(new Dimension(dimension.width, 100));
    }
}
```

El resultado de este código (separator.java en el CD) se muestra en la figura 14.9. Como muestra la figura, el separador aparece entre los dos campos de texto.

A pesar de todo, tenemos un problema, ¿qué sucede si cambia el tamaño del applet? En ese caso, podríamos querer cambiar el tamaño del separador para que se ajuste. Podemos hacerlo procesando los eventos de cambio de tamaño; consulte el siguiente tema para ver los detalles.

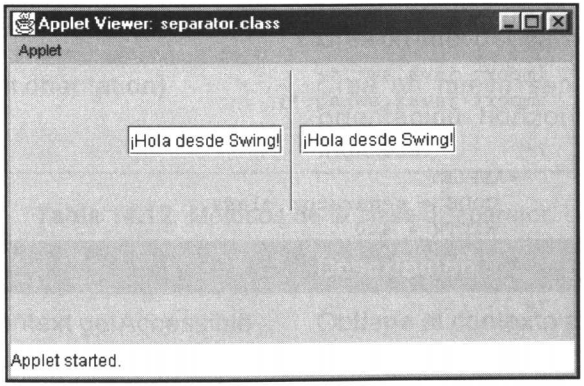


Figura 14.9. Creación de un separador.

Cambio de tamaño automático de separadores

Para cambiar el tamaño de un separador, podemos capturar los eventos de cambio de tamaño del componente utilizando la interfaz ComponentListener, que maneja los eventos de cambio de tamaño para componentes, incluyendo applet y ventanas. Los métodos de esta interfaz se muestran en la tabla 14.13.

Tabla 14.13. Métodos de la interfaz ComponentListener.

Método	Hace ésto
void componentHidden(Component-Event e)	Llamado cuando se hace invisible el componente.
void componentMoved(Component-Event e)	Llamado cuando la posición del componente cambia.
void componentResized(Component-Event e)	Llamado cuando el tamaño del componente cambia.
void componentShown(Component-Event e)	Llamado cuando se hace visible el componente.

Por ejemplo, cuando cambia el tamaño de su applet, puede cambiar el tamaño del separador asignando el que desee. Aquí tenemos un ejemplo en el que creamos un separador entre los campos de texto, que va desde la parte superior de la ventana del área de cliente del applet hasta la parte inferior y mantiene esa extensión incluso cuando cambia de tamaño el applet. Aquí tenemos el código: observe el código de cambio de tamaño en los métodos componentShown (para visualizar por primera vez el separador) y componentResized:

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

/*
  <APPLET
    CODE = separatorevents.class
    WIDTH = 400
    HEIGHT = 200 >
  </APPLET>
*/

public class separatorevents extends JApplet implements ComponentListener
{
    JSeparator jseparator = new JSeparator(JSeparator.VERTICAL);
```

```
    Dimension dimension = jseparator.getPreferredSize();

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(new JTextField("¡Hola desde Swing!"));
        contentPane.add(jseparator);
        contentPane.add(new JTextField("¡Hola desde Swing!"));

        addComponentListener(this);
    }

    public void componentShown(ComponentEvent e)
    {
        jseparator.setPreferredSize(new Dimension(dimension.width,
            getSize().height));
        jseparator.revalidate();
    }

    public void componentResized(ComponentEvent e)
    {
        jseparator.setPreferredSize(new Dimension(dimension.width,
            getSize().height));
        jseparator.revalidate();
    }

    public void componentMoved(ComponentEvent e) {}
    public void componentHidden(ComponentEvent e) {}
}
```

El resultado de este código se muestra en la figura 14.10. Como puede ver, el separador se extiende desde la parte superior del área de cliente del applet hasta la parte inferior y cambiará de tamaño automáticamente junto con el applet. Este ejemplo se encuentra en el archivo separatorevents.java del CD.

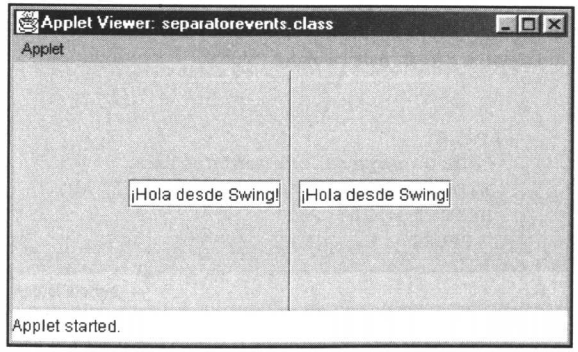


Figura 14.10. Creación de un separador que cambia de tamaño automáticamente.

Creación de un selector de color

"Ahora tenemos un problema real", dice el programador novato. "Quiero permitir a los usuarios la selección de un color de dibujo para mi nuevo programa, *SuperDuperArtisticPro*, pero no puedo pedirles que introduzcan los nuevos valores de color". "¿Por qué no utiliza un selector de color?", preguntamos. "¡Buena idea!", dice el PN. "Escriba el código y yo lo miro".

Podemos permitir al usuario seleccionar un color con la clase JColorChooser de la Swing, que es un cuadro de diálogo ya creado que presenta al usuario diversas formas de seleccionar colores. Aquí tenemos el diagrama de herencia de la clase JColorChooser:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JColorChooser
```

Puede encontrar los campos de la clase JColorChooser en la tabla 14.14, la tabla 14.15 muestra sus constructores y la tabla 14.16 muestra sus métodos.

Tabla 14.14. Campos de la clase JColorChooser.

Campo	Descripción
protected AccessibleContext accessibleContext	El contexto accesible.
static String CHOOSER_PANELS_PROPERTY	El nombre de la propiedad de la matriz del panel de selector.
static String PREVIEW_PANEL_PROPERTY	El nombre la propiedad del panel de vista previa.
static String SELECTION_MODEL_PROPERTY	El nombre de la propiedad del modelo de selección.

Tabla 14.15. Constructores de la clase JColorChooser.

Constructor	Descripción
JColorChooser()	Construye un panel selector de color.

Constructor	Descripción
JColorChooser(Color initialColor)	Construye un panel selector de color con el color inicial indicado.
JColorChooser(ColorSelectionModel model)	Construye un panel selector de color con el modelo de selección de color indicado.

Tabla 14.16. Métodos de la clase JColorChooser.

Método	Descripción
void addChooserPanel(AbstractColorChooserPanel panel)	Añade un panel selector de color.
static JDialog createDialog(Component c, String title, boolean modal, JColorChooser chooserPane, ActionListener okListener, ActionListener cancelListener)	Construye y devuelve un nuevo diálogo con el color y indicado por el selector de color con los botones Aceptar, Cancelar y Reiniciar.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible.
AbstractColorChooserPanel[] getChooserPanels()	Obtiene los paneles de color indicados.
Color getColor()	Obtiene el valor actual del color a partir del selector de color.
JComponent getPreviewPanel()	Obtiene el panel de vista previa que muestra un color seleccionado.
ColorSelectionModel getSelectionModel()	Obtiene el modelo de datos que maneja selecciones de color.
ColorChooserUI getUI()	Obtiene el objeto apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza este componente.
protected String paramString()	Obtiene una representación de cadena de este objeto JColorChooser.
AbstractColorChooserPanel removeChooserPanel(AbstractColorChooserPanel panel)	Elimina el panel de color indicado.

Método	Descripción
void setChooserPanels(AbstractColorChooserPanel[] panels)	Especifica los paneles de color utilizados para seleccionar un valor de color.
void setColor(Color color)	Asigna al color actual de selector de color el color indicado.
void setColor(int c)	Asigna al color actual del selector de color el color indicado.
void setColor(int r, int g, int b)	Asigna al color actual del sector de color al color RGB indicado.
void setPreviewPanel(JComponent preview)	Asigna el panel de vista previa actual.
void setSelectionModel(ColorSelectionModel newModel)	Asigna el modelo que contiene el color seleccionado.
void setUI(ColorChooserUI ui)	Asigna al objeto apariencia que renderiza este componente.
static Color showDialog(Component component, String title, Color initialColor)	Muestra el cuadro de diálogo de selección de color modal.
void updateUI()	Llamada por la clase UIManager cuando la apariencia ha cambiado.

Es sencillo utilizar un sector de color; lo único que tenemos que hacer es mostrarlo con el método showDialog y después pasar a este método un objeto padre, un título y un color predeterminado. Este método devuelve el color seleccionado por el usuario como un objeto Color (o el color predeterminado si el usuario no realizó selección alguna).

Vamos a examinar un ejemplo en el que situamos un botón en un panel y permitimos al usuario visualizar el selector de color cuando haga clic sobre el botón. Una vez que el usuario seleccione un color, el código cambia el fondo del panel a ese color. Aquí tenemos el código:

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

/*
  <APPLET
    CODE = colorchooser.class
    WIDTH = 200
    HEIGHT = 200 >
```

```
  </APPLET>
*/

public class colorchooser extends JApplet implements ActionListener
{
    JPanel jpanel = new JPanel();
    JButton jbutton;

    public void init()
    {
      jbutton = new JButton("Haga clic aquí para cambiar los colores.");
        jbutton.addActionListener(this);
        jpanel.add(jbutton);
        getContentPane().add(jpanel); //, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e)
    {
        Color color = JColorChooser.showDialog(colorchooser.this,
            "Seleccione un nuevo color...", Color.white);

        jpanel.setBackground(color);
    }
}
```

Como puede ver, es sencillo utilizar un selector de color; éste apareceen la figura 14.11. Cuando el usuario selecciona un color en el selector, el nuevo color aparece en el panel del applet, como se muestra en la figura 14.12. Este ejemplo (colorchooser.java en el CD) es un éxito.

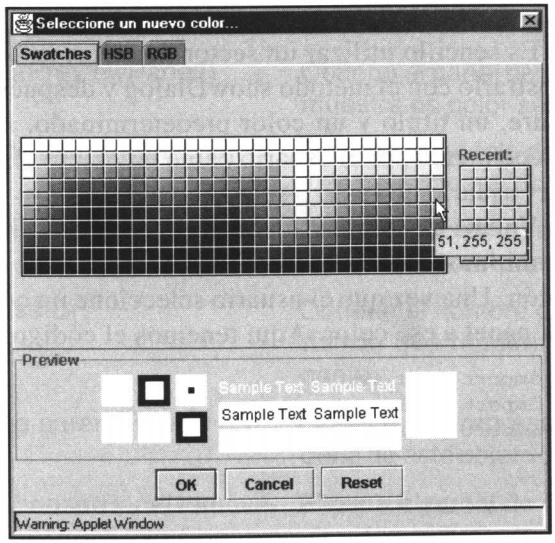


Figura 14.11. Uso de un selector de color.

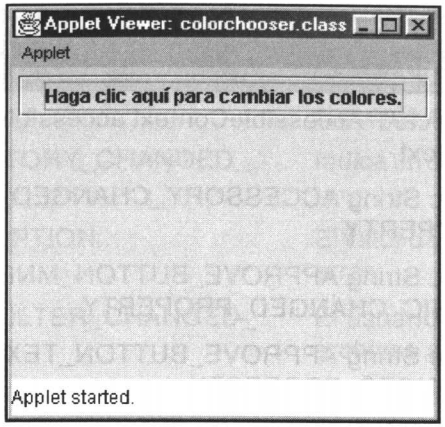


Figura 14.12. Cambio del color de un panel a un color seleccionado.

Creación de selectores de archivos

"Ajá", dice el programador novato, "necesito obtener el nombre de un archivo de usuario; ¿existe alguna forma rápida de hacerlo?" "Claro", decimos, "un campo de texto". "Bien", dice el PN, "quiero permitir que el usuario explore el disco para obtener el archivo correcto si fuera necesario". "Ah", decimos, "entonces desea un selector de archivos".

Los selectores de archivos son cuadros de diálogo que permiten al usuario especificar el nombre y vía de acceso de un archivo, permitiéndole explorar el espacio de disco si es necesario. Los selectores de archivos quedan soportados por la clase JFileChooser. Aquí tenemos el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JFileChooser
```

La tabla 14.17 muestra los campos de la clase JFileChooser, la tabla 14.18 sus constructores y la tabla 14.19 sus métodos.

Tabla 14.17. Campos de la clase JFileChooser.

Campo	Descripción
protected AccessibleContext accessible-Context	El contexto accesible.
static String ACCESSORY_CHANGED_ PROPERTY	Indica que un componente accesorio diferente está en uso.
static String APPROVE_BUTTON_MNE-MONIC_CHANGED_PROPERTY	Indica un cambio en el mnemónico para el botón de Aceptar.
static String APPROVE_BUTTON_TEXT_ CHANGED_PROPERTY	Indica un cambio en el texto del botón Aceptar/Sí/Aprobar.
static String APPROVE_BUTTON_TOOL _TIP_TEXT_CHANGED_PROPERTY	Indica un cambio en el texto de la ayuda de herramientas para el botón Aceptar/Sí/Aprobar.
static int APPROVE_OPTION	El valor de retorno si se selecciona el botón Aceptar/Sí/ Aprobar.
static String APPROVE_SELECTION	Instrucción para probar la selección actual.
static int CANCEL_OPTION	El valor de retorno si selecciona el botón Cancelar.
static String CANCEL_SELECTION	Instrucción para cancelar la selección actual.
static String CHOOSABLE_FILE_FILTER_ CHANGED_PROPERTY	Indica un cambio en la lista de títulos de archivo predeterminados entre los que puede seleccionar el usuario.
static int CUSTOM_DIALOG	Valor de tipo indicando que el selector de activos soporta una operación archivo especificada por el desarrollador.
static String DIALOG_TITLE_CHANGED_ PROPERTY	Indica un cambio en el título del cuadro de diálogo.
static String DIALOG_TYPE_CHANGED_ PROPERTY	Indica un cambio en el tipo de archivos mostrados (únicamente archivos, sólo directorios o tanto archivos como directorios).

Campo	Descripción
static int DIRECTORIES_ONLY	Instrucción para visualizar únicamente directorios.
static String DIRECTORY_CHANGED_PROPERTY	Indica un cambio de directorio.
static int ERROR_OPTION	El valor de retorno si ocurre un error.
static String FILE_FILTER_CHANGED_PROPERTY	El usuario cambia el tipo de archivos a mostrar.
static String FILE_HIDING_CHANGED_PROPERTY	Indica un cambio en la propiedad "Mostrar archivos ocultos".
static String FILE_SELECTION_MODE_CHANGED_PROPERTY	Indica un cambio en laclase de selección (simple, múltiple y así sucesivamente).
static String FILE_SYSTEM_VIEW_CHANGED_PROPERTY	Indica que se está utilizando un objeto diferente para encontrar los archivos disponibles en el sistema.
static String FILE_VIEW_CHANGED_PROPERTY	Indica que se utiliza un objeto distinto para recuperar información de los archivos.
static int FILES_AND_DIRECTORIES	Instrucción para visualizar tanto archivos como directorios.
static int FILES_ONLY	Instrucción para visualizar únicamente archivos.
static String MULTI_SELECTION_ENABLED_CHANGED_PROPERTY	Permite selección múltiple de archivos.
static int OPEN_DIALOG	Indica que el selector de archivos soporta una operación de apertura de archivos.
static int SAVE_DIALOG	Indica que el selector de archivos soporta la operación para guardar archivos.
static String SELECTED_FILE_CHANGED_PROPERTY	Indica un cambio en la selección única de archivos del usuario.

Campo	Descripción
static String SELECTED_FILES_CHANGED_PROPERTY	Indica un cambio en la selección múltiple de archivos del usuario.

Tabla 14.18. Constructores de la clase JFileChooser.

Constructor	Descripción
JFileChooser()	Construye un objeto JFileChooser.
JFileChooser(File currentDirectory)	Construye un objeto JFileChooser utilizando un archivo dado como vía de acceso.
JFileChooser(File currentDirectory, FileSystemView fsv)	Construye un objeto JFileChooser utilizando el directorio actual dado y la vista del sistema archivos.
JFileChooser(FileSystemView fsv)	Construye un objeto JFileChooser utilizando la vista del sistema archivos dada.
JFileChooser(String currentDirectoryPath)	Construye un objeto JFileChooser utilizando la vía de acceso dada.
JFileChooser(String currentDirectoryPath, FileSystemView fsv)	Construye un objeto JFileChooser utilizando el directorio actual en curso como vía de acceso y la vista del sistema archivos.

Tabla 14.19. Métodos de la clase JFileChooser.

Método	Descripción
boolean accept(File f)	Devuelve cierto si el archivo deber mostrarse.
void addActionListener(ActionListener l)	Añade un receptor de acciones.
void addChoosableFileFilter(FileFilter filter)	Añade un filtro a la lista de filtros de archivo seleccionables.

Método	Descripción
void approveSelection()	Llamado por la IU cuando el usuario hace clic sobre los botones Guardar o Abrir.
void cancelSelection()	Llamada por la IU cuando el usuario hace clic sobre el botón Cancelar.
void changeToParentDirectory()	Asigna el padre del directorio actual.
void ensureFileIsVisible(File f)	Asegura que el campo indicado es visible.
protected void fireActionPerformed(String command)	Notifica a todos los receptores que han registrado interés por la notificación de este tipo de evento.
FileFilter getAcceptAllFileFilter()	Obtiene el filtro archivos AcceptAll.
AccessibleContext getAccessibleContext()	Obtiene el contexto accesible asociado con este objeto JFile-Chooser.
JComponent getAccessory()	Obtiene el componente accesorio.
int getApproveButtonMnemonic()	Obtiene el mnemónico del botón Aprobar.
String getApproveButtonText()	Obtiene el texto usado en el botón aprobar en FileChooser-UI.
String getApproveButtonToolTipText()	Obtiene el texto de la ayuda de herramientas utilizada en el botón Aprobar.
FileFilter[] getChoosableFileFilters()	Obtiene la lista de filtros de archivo seleccionados por el usuario.
File getCurrentDirectory()	Devuelve el directorio actual.
String getDescription(File f)	Obtiene la descripción de archivo.

Método	Descripción
String getDialogTitle()	Obtiene la cadena que va en la barra de título del selector de archivos.
int getDialogType()	Obtiene el tipo de este cuadro de diálogo.
FileFilter getFileFilter()	Obtiene el filtro de archivo seleccionado en curso.
int getFileSelectionMode()	Obtiene el modo de selección de archivos actual.
FileSystemView getFileSystemView()	Obtiene la vista del sistema archivos.
FileView getFileView()	Obtiene la vista del archivo actual.
Icon getIcon(File f)	Obtiene el icono para este archivo o tipo de archivo, dependiendo del sistema.
String getName(File f)	Obtiene el nombre de archivo.
File getSelectedFile()	Obtiene elarchivo seleccionado.
File[] getSelectedFiles()	Obtiene una lista de los activos seleccionados si el selector de archivos permite la selección múltiple.
String getTypeDescription(File f)	Obtiene el tipo de archivo.
FileChooserUI getUI()	Obtiene el objeto IU que implementa la apariencia de este componente.
String getUIClassID()	Obtiene una cadena que especifica el nombre de la clase de apariencia que renderiza este componente.
boolean isDirectorySelectionEnabled()	Un método conveniente que determina si los directorios se pueden seleccionar.
boolean isFileHidingEnabled()	Devuelve cierto si los activos ocultos no se muestran en el selector de archivos.

Método	Descripción
boolean isFileSelectionEnabled()	Un método conveniente que determina si los archivos son seleccionables basado en el modo de selección de archivo actual.
boolean isMultiSelectionEnabled()	Devuelve cierto si se pueden seleccionar múltiples archivos.
boolean isTraversable(File f)	Devuelve cierto si el archivo (directorio) se puede visitar.
protected String paramString()	Obtiene una representación de cadena de este objeto JFile-Chooser.
void removeActionListener(ActionListener l)	Elimina un receptor de acciones del botón.
boolean removeChoosableFileFilter(File-lisFilter f)	Elimina un filtro a partir de la lista de filtros de archivos seleccionables.
void rescanCurrentDirectory()	Explorar de nuevo la lista de archivos a partir del directorio actual.
void resetChoosableFileFilters()	Reinicia el filtro de archivo seleccionado de la lista a su estado inicial.
void setAccessory(JComponent new-Accessory)	Asigna un componente accesorio.
void setApproveButtonMnemonic(char mnemonic)	Asigna el mnemónico del botón Aprobar utilizando un carácter.
void setApproveButtonMnemonic(int mnemonic)	Asigna mnemónico del botón Aprobar utilizando un código de teclado numérico.
void setApproveButtonText(String approve-ButtonText)	Asigna el texto utilizado en el botón Aprobar.
void setApproveButtonToolTipText(String toolTipText)	Asigna al texto de la ayuda de herramientas utilizada en el botón Aprobar.
void setCurrentDirectory(File dir)	Asigna el directorio actual.

Método	Descripción
void setDialogTitle(String dialogTitle)	Asigna la cadena que va en la barra de título de la ventana del selector de archivos.
void setDialogType(int dialogType)	Asigna el tipo de este cuadro de diálogo.
void setFileFilter(FileFilter filter)	Asigna el filtro de archivos actual.
void setFileHidingEnabled(boolean b)	Asigna la ocultación de archivos.
void setFileSelectionMode(int mode)	Asigna el selector de archivos que permite al usuario seleccionar únicamente archivos, únicamente directorios o ambos.
void setFileSystemView(FileSystemView fsv)	Asigna la vista del sistema archivos que utiliza un objeto JFileChooser.
void setFileView(FileView fileView)	Asigna la vista de archivos utilizada para obtener información de la IU, como el icono que representará el archivo o la descripción del tipo de un archivo.
void setMultiSelectionEnabled(boolean b)	Configura el selector de archivos para que pueda realizar selección múltiple.
void setSelectedFile(File selectedFile)	Asigna el archivo seleccionado.
void setSelectedFiles(File[] selectedFiles)	Asigna la lista de archivos seleccionados si el selector de archivos está configurado para permitir selección múltiple.
protected void setup(FileSystemView view)	Realiza la configuración e inicialización del constructor.
int showDialog(Component parent, String approveButtonText)	Visualiza un cuadro de diálogo de selector de archivos personalizado con un botón Aprobar personalizado.

Método	Descripción
int showOpenDialog(Component parent)	Visualiza un cuadro de diálogo de selección de archivos "Abrir archivo".
int showSaveDialog(Component parent)	Visualiza un cuadro de diálogo de selector de archivos "Guardar archivo".
void updateUI()	Llamada por la clase UIFactory cuando cambia la apariencia.

Puede utilizar el método showOpenDialog de la clase JFileChooser con el fin de mostrar un selector de activos para buscar archivos que necesite abrir, y puede utilizar el método showSaveDialog para mostrar un selector de archivos con el fin de especificar el nombre de archivo y vía de acceso que desea utilizar para guardar un archivo. Estos métodos devuelven los siguientes valores:

- ***APPROVE_OPTION***. Se devuelve si el usuario hace clic sobre un botón de aprobación, Guardar o Abrir.
- ***CANCEL_OPTION***. Se devuelve si el usuario hace clic sobre Cancelar.
- ***ERROR_OPTION***. Se devuelve si existe un error.

Puede obtener el archivo seleccionado como un objeto File con el método getSelectedFile del selector de archivos (examinaremos la clase File en este libro más tarde) y puede utilizar los métodos getPath, getName y otros de la clase File para devolver información sobre el archivo.

Vamos a examinar un ejemplo que hace funcionar todo esto. En este caso, visualizaremos un selector de apertura de archivos cuando el usuario hace clic sobre un botón. Permitimos que el usuario seleccione un archivo y después visualizaremos el nombre del archivo y su vía de acceso en un campo de texto. Debido a que normalmente no abrimos archivos desde los applet (por razones de seguridad), haremos que este ejemplo sea una aplicación. Aquí tenemos el código (observe que comprobamos el valor de retorno de showOpenDialog para ver si el usuario ha hecho clic sobre el botón Abrir antes de visualizar el nombre de archivo en el campo):

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

```
import javax.swing.filechooser.*;

public class filechooser extends JFrame implements ActionListener
{
    JFileChooser chooser = new JFileChooser();
    JButton jbutton = new JButton("Mostrar selector de archivos");
    JTextField jtextfield = new JTextField(30);

    public filechooser()
    {
        super();
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jbutton);
        contentPane.add(jtextfield);

        jbutton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        int result = chooser.showOpenDialog(null);
        File fileobj = chooser.getSelectedFile();

        if(result == JFileChooser.APPROVE_OPTION) {
            jtextfield.setText("Su selección " + fileobj.getPath());
        } else if(result == JFileChooser.CANCEL_OPTION) {
            jtextfield.setText("Hizo clic sobre Cancelar");
        }
    }

    public static void main(String args[])
    {
        JFrame f = new filechooser();
        f.setBounds(200, 200, 400, 200);

        f.setVisible(true);

        f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}
```

La figura 14.13 muestra el selector de archivos que visualiza esta aplicación. El usuario puede explorar y seleccionar un archivo. Cuando lo hace, ya sea resaltando un archivo, haciendo clic sobre el botón Abrir en el selector de archivos o simplemente haciendo doble clic sobre el archivo en el selector de

archivos, se cierra el selector de archivos y aparecen el nombre y vía de acceso del archivo seleccionado en el campo de texto de la aplicación, como se muestra en la figura 14.14. Eso es todo. Este ejemplo se encuentra en el archivo filechooser.java del CD.

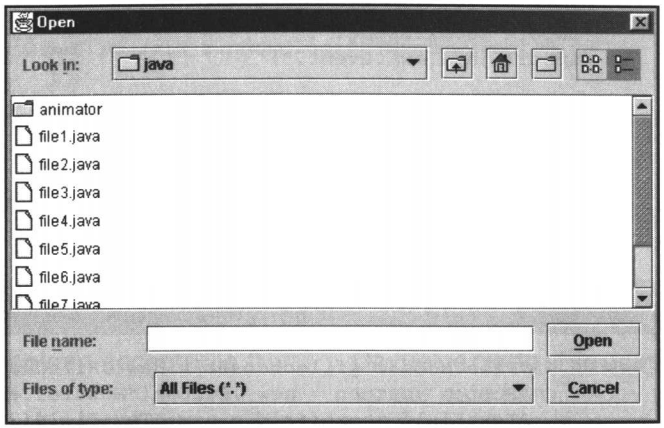


Figura 14.13. Creación de un selector de archivos.

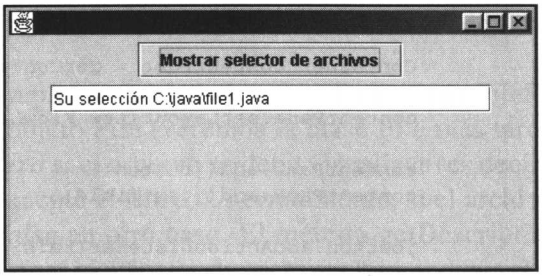


Figura 14.14. Determinación del archivo seleccionado.

Creación de filtros para selectores de archivo

Existen muchas formas de personalizar los electores de archivo y una de ellas es crear *filtros* de archivos. Puede utilizar filtros para asignar los tipos de archivos (basándose en su extensión) que mostrará un selector de archivos. Los filtros de archivos derivan de la clase FileFilter. Para añadir un filtro de archivos a un selector, utilice el método addChoosableFileFilter de JFileChooser.

Vamos a examinar un ejemplo en el que añadimos dos nuevos filtros de archivo a un selector; un filtro para archivos .GIF y otro para archivos .Java.

Estos dos filtros están soportados por las clases filter1 y filter2, por lo que podemos utilizar el método addChoosableFileFilter para añadirlos a un selector de archivos, como se realiza a continuación:

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

/*
  <APPLET
    CODE = filechooserfilter.class
    WIDTH = 200
    HEIGHT = 200 >
  </APPLET>
*/

public class filechooserfilter extends JFrame implements ActionListener
{
    JFileChooser jfilechooser = new JFileChooser();
    JButton jbutton = new JButton("Mostrar el selector de archivos");
    JTextField jtextfield = new JTextField(20);

    public filechooserfilter()
    {
        super();

        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jbutton);
        contentPane.add(jtextfield);

        jbutton.addActionListener(this);

        jfilechooser.addChoosableFileFilter(new filter1());
        jfilechooser.addChoosableFileFilter(new filter2());
    }

    public void actionPerformed(ActionEvent e)
    {
        int result = jfilechooser.showOpenDialog(null);

        if(result == JFileChooser.APPROVE_OPTION) {
            jtextfield.setText("Su selección es " +
                jfilechooser.getSelectedFile().getPath());
        }
    }

    public static void main(String a[])
    {
        JFrame jframe = new filechooserfilter();
```

```
        jframe.setBounds(200, 200, 300, 300);

        jframe.setVisible(true);

        jframe.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);

        jframe.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}
```

Tabla 14.20. Métodos de la clase FileFilter.

Método	Descripción
abstract boolean accept(File f)	Devuelve cierto si se acepta el archivo por este filtro.
abstract String getDescription()	Devuelve la descripción de este filtro.

Ahora hemos creado las clases filter1 y filter2. Estas clases derivan de la clase abstracta FileFilter y encontraremos los métodos de esta claseen la tabla 14.20.

No es difícil implementar los métodos de la clase FileFilter; el método *accept* recibe un objeto File (veremos la clase File más tarde en este libro). Devuelve *verdadero* si el archivo se debe visualizar (es decir, si su extensión es una de las que acepta el filtro o, normalmente, si el archivo corresponde a un directorio) y *false* en otro caso. El método getDescription devuelve una cadena que visualizará el selector de archivos para indicar al usuario los tipos de archivos para los que vale el filtro. La clase filter1 aceptará archivos .gif (descritos como "Archivos GIF (*.gif) " en este filtro) y directorios, y la clase filter2 filtra archivos .Java (descritos como "Archivos Java (*.Java)" en este filtro) y directorios. Aquí tenemos el código de estas clases:

```
class filter1 extends javax.swing.filechooser.FileFilter
{
    public boolean accept(File fileobj)
    {
        String extension = "";

        if(fileobj.getPath().lastIndexOf('.') > 0)
            extension = fileobj.getPath().substring(
                fileobj.getPath().lastIndexOf('.')
                + 1).toLowerCase();

        if(extension != "")
```

```
            return extension.equals("gif");
        else
            return fileobj.isDirectory();
    }

    public String getDescription()
    {
        return "Archivos Gif (*.gif)";
    }
}

class filter2 extends javax.swing.filechooser.FileFilter
{
    public boolean accept(File fileobj)
    {
        String extension = "";

        if(fileobj.getPath().lastIndexOf('.') > 0)
            extension = fileobj.getPath().substring(
                fileobj.getPath().lastIndexOf('.')
                + 1).toLowerCase();

        if(extension != "")
            return extension.equals("java");
        else
            return fileobj.isDirectory();
    }

    public String getDescription()
    {
        return "Archivos Java (*.java)";
    }
}
```

Y eso es todo. El resultado de este código se muestra en la figura 14.15, donde puede ver el filtro .java en acción. El uso de filtros como este, puede facilitar al usuario la búsqueda de archivos de un tipo particular. Este ejemplo es un éxito y puede encontrar el archivo filechooserfilter.java en el CD.

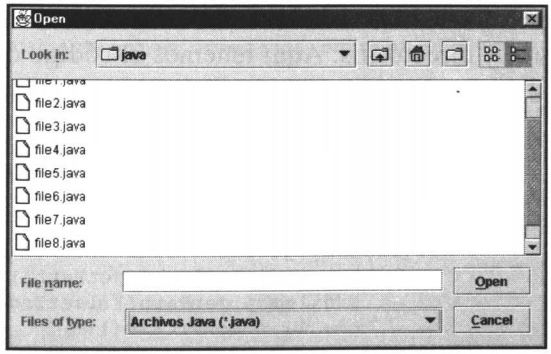


Figura 14.15. Uso de los filtros en un selector de filtros.

15 Swing: paneles de capas, de lengüetas, separadores y distribuciones

En este capítulo, examinaremos algunos de los contenedores de poco peso de la Swing: paneles de capas, paneles de lengüetas y paneles de separación. La Swing proporciona varios continentes de poco peso, incluyendo JPanel (que ya se ha visto), para permitir a los programadores manejar componentes de forma sencilla. Los paneles de capas, introducidos en el capítulo 11, nos permiten especificar la *capa* en los componentes que añadimos, facilitando la configuración del *orden Z*, o fijando el orden en el eje Z (que apunta hacia el exterior de la pantalla) de los componentes. Los paneles de lengüetas nos permiten ordenar los componentes como si los añadiéramos a una colección de archivos de carpeta de lengüetas, y podemos hacer clic sobre las lengüetas para abrir cada carpeta. Finalmente los paneles de separación no permiten visualizar dos componentes colindantes y ajustar la cantidad visible de cada uno; una técnica usada habitualmente para soportar dos vistas del mismo modelo. También examinaremos la distribución en este capítulo. Primero vimos la distribución del AWT en el capítulo 7. La Swing soporta todas estas distribuciones y dos más: distribución de cuadro y distribución de superposición. Examinaremos estas dos nuevas distribuciones en este capítulo. La distribución de cuadro nos permite crear filas de columnas de componentes.

La distribución superpuesta, como implica el nombre, nos permite dibujar componentes superpuestos. La Swing también define la clase Box, que nos permite distribuir componentes utilizando estructuras visuales, cruces, áreas rígidas y pegadas. Veremos todo en este capítulo.

Paneles de capas

Los paneles de capas, contenedores de poco peso, son paneles importantes para los contenedores de alto peso de la Swing como JApplet y JFrame. Los paneles de capas se dividen en varias capas verticales que pueden trabajar conjuntamente con la Swing. Esta es una de las áreas en las que la implementación de la Swing sobre AWT no es transparente para el programador, debido a que puede ver las capas que utiliza la Swing para visualizar los cuadros de diálogo, arrastrar componentes, menús emergentes y otros. Uno de los aspectos más populares del panel de capas es que es la raíz del panel de contención.

Paneles de lengüetas

Los cuadros de diálogo que permiten a los usuarios seleccionar entre muchas opciones han cambiado a medida que los programas se han hecho más complejos y ofrecían más opciones. Se implementan personalizadamente de esta forma, organizando la configuración del programa en lengüetas para la pantalla, información de usuario, opciones generales, archivos y otras. La Swing soporta ahora paneles de lengüetas para permitirle crear cuadros de diálogo y programas de esta clase. Como es de esperar, todas las capacidades de la Swing están disponibles aquí, como la visualización de imágenes de lengüetas.

Paneles de separación

Otra técnica popular de programación de las IU hoy día es permitir al usuario dividir una vista en un modelo de datos, creando de esta forma una nueva vista con los datos. Los procesadores de texto a veces permiten a los usuarios dividir sus presentaciones en dos paneles de tal forma que puedan pasar de forma independiente entre dos áreas de un documento. La Swing soporta los paneles de separación para permitir a los programas presentar dos

componentes conjuntamente. Estos componentes pueden representar vistas del mismo u otro modelo de datos.

Distribución

En el capítulo 7 examinamos la distribución del AWT, y la Swing soporta estas distribuciones así como dos más: distribución de cuadro y distribución superpuesta. Examinaremos estas dos nuevas distribuciones en este capítulo. De las dos, la distribución de cuadro es la más popular, debido a que permite distribuir los componentes en filas y columnas horizontales y verticales, llamados *cuadros*. De hecho, la Swing también soporta una clase Box que va más allá, permitiendo especificar el espacio de distribución de componentes, como examinaremos. El gestor de distribución de superposición nos permite superponer componentes de una forma bien definida y, aunque no es tan común como los otros sectores de distribución, tiene su utilidad.

Es suficiente para el resumen de lo que aparece en este capítulo. Como puede ver, vendrá mucho más. Es el momento de pasar a la sección inicial, comenzando con una descripción de los componentes de desplazamiento de la Swing.

Comprensión de los componentes de la Swing y el orden Z

"Java ha fallado de nuevo", dice el programador novato. "Deposité algunos componentes en la misma zona de mi programa por error y aparecían unos encima de otros". Sonreímos y decimos, "Es perfectamente posible. De hecho, en la Swing no está fuera de lo común superponer componentes". El PN dice, "¡Vaya!"

Cuando añadimos componentes de poco peso a un panel contenedor, los componentes se dibujarán sencillamente sobre el panel y nada nos impide hacer que los componentes se superpongan. Es decir, siempre y cuando no utilicemos un gestor de distribución (excepto para el gestor de distribución de superposición) que impediría la superposición. Cuando los componentes se superponen, su orden Z pasa a ser muy importante; el *orden Z* representa el emplazamiento relativo de los componentes a lo largo del eje Z, que sale fuera de la pantalla. El componente de poco peso que se haya añadido primero aparecerá por encima de aquellos que se añadan posteriormente. Puede

también añadir componentes pesados de la AWT a su programa y esos componentes, que tienen su propio sistema operativo de ventanas, normalmente aparecerán por encima de los componentes de la Swing.

Aquí tenemos un ejemplo en el cual eliminamos el gestor de distribución de borde predeterminado de un panel de contenidos y añadimos una serie de etiquetas superpuestas, mostrando de esta forma que la primera etiqueta añadida aparecerá por encima del resto. Aquí tenemos el código (observe que hemos añadido un borde a las etiquetas para hacer visible la superposición):

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

/*
  <APPLET
    CODE = layered.class
    WIDTH = 350
    HEIGHT = 280 >
  </APPLET>
*/

public class layered extends JApplet
{
    JLabel labels[];

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(null);

        labels = new JLabel[6];

        labels[0] = new JLabel("Etiqueta 0");
        labels[0].setOpaque(true);
        labels[0].setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(labels[0]);

        labels[1] = new JLabel("Etiqueta 1");
        labels[1].setOpaque(true);
        labels[1].setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(labels[1]);

        labels[2] = new JLabel("Etiqueta 2");
        labels[2].setOpaque(true);
        labels[2].setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(labels[2]);

        labels[3] = new JLabel("Etiqueta 3");
        labels[3].setOpaque(true);
        labels[3].setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(labels[3]);

        labels[4] = new JLabel("Etiqueta 4");
```

```
            labels[4].setOpaque(true);
            labels[4].setBorder(BorderFactory.createEtchedBorder());
            contentPane.add(labels[4]);

            labels[5] = new JLabel("Etiqueta 5");
            labels[5].setOpaque(true);
            labels[5].setBorder(BorderFactory.createEtchedBorder());
            contentPane.add(labels[5]);

            for(int loop_index = 0; loop_index < 6; loop_index++) {
             labels[loop_index].setBounds(40 * loop_index, 40 * loop_index,
100, 60);
            }
        }
    }
```

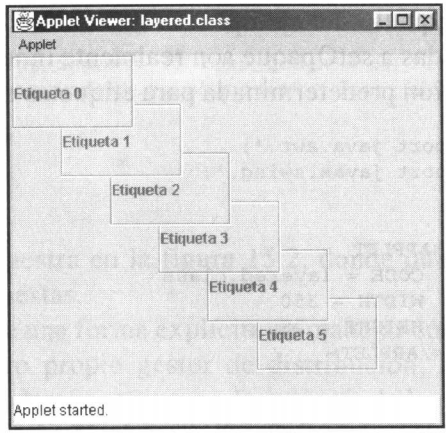


Figura 15.1. Controles superpuestos.

El resultado se muestra en la figura 15.1. Como ve, la primera etiqueta añadida aparece por encima de las otras. La siguiente etiqueta añadida aparece por encima de las posteriores y así sucesivamente, definiendo de esta forma un orden Z distinto para cada etiqueta. Este ejemplo se encuentra en el archivo layered.java del CD.

Debe también observar que hemos utilizado el método setOpaque para cada etiqueta con el fin de asegurar que las etiquetas que quedan por debajo no se muestren de ninguna forma; consulte el siguiente tema para obtener más detalles.

Transparencia en los componentes de la Swing

"Acabo de ver el programa más extraño", dice el programador novato. "Realmente podía ver todos los controles como si fueran transparentes". "No

es extraño", decimos. "De hecho, muchos programas de la Swing lo hacen de forma intencionada". "¡Dígame más!" Dice el PN.

Debido a que los controles de poco peso simplemente se dibujan dentro de sus contenedores, podemos hacerlos transparentes. Cuando hacemos un control transparente, no pinta su fondo cuando lo dibujamos o repintamos, de tal forma que los controles por debajo se pueden ver. Esta es una técnica ampliamente utilizada que nos permite hacer que su apariencia sea como si dibujáramos componentes que fueran irregulares y no rectangulares.

En el tema anterior, escribimos un ejemplo en el que mostramos etiquetas opacas superpuestas de la Swing, pero también explícitamente podemos hacerlas transparentes. Puede asignar la transparencia de los componentes de la Swing con el método setOpaque. A continuación mostramos cómo hacer que las etiquetas del ejemplo del tema anterior sean transparentes (de hecho, las llamadas a setOpaque son realmente innecesarias aquí debido a que la configuración predeterminada para etiquetas es hacerlas transparentes):

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

/*
  <APPLET
    CODE = layered.class
    WIDTH = 350
    HEIGHT = 280 >
  </APPLET>
*/

public class layered extends JApplet
{
    JLabel labels[];

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(null);

        labels = new JLabel[6];

        labels[0] = new JLabel("Etiqueta 0");
        labels[0].setOpaque(false);
        labels[0].setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(labels[0]);

        labels[1] = new JLabel("Etiqueta 1");
        labels[1].setOpaque(false);
        labels[1].setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(labels[1]);

        labels[2] = new JLabel("Etiqueta 2");
        labels[2].setOpaque(false);
```

```
            labels[2].setBorder(BorderFactory.createEtchedBorder());
            contentPane.add(labels[2]);

            labels[3] = new JLabel("Etiqueta 3");
            labels[3].setOpaque(false);
            labels[3].setBorder(BorderFactory.createEtchedBorder());
            contentPane.add(labels[3]);

            labels[4] = new JLabel("Etiqueta 4");
            labels[4].setOpaque(false);
            labels[4].setBorder(BorderFactory.createEtchedBorder());
            contentPane.add(labels[4]);

            labels[5] = new JLabel("Etiqueta 5");
            labels[5].setOpaque(false);
            labels[5].setBorder(BorderFactory.createEtchedBorder());
            contentPane.add(labels[5]);

            for(int loop_index = 0; loop_index < 6; loop_index++) {
             labels[loop_index].setBounds(40 * loop_index, 40 * loop_index,
100, 60);
            }
        }
    }
```

El resultado se muestra en la figura 15.2, donde puede ver las etiquetas transparentes superpuestas.

Observe que existe una forma explícita de manejar controles superpuestos sin configurar nuestro propio gestor de distribución. Esto involucra a un panel de capas que podemos utilizar en la mayoría de los programas. Consulte el siguiente tema para obtener más detalles.

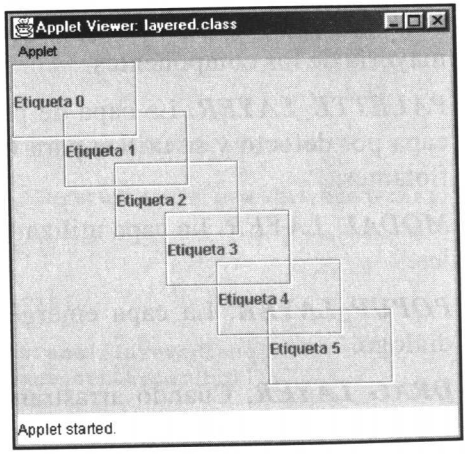


Figura 15.2. Transparencia de controles.

Uso de paneles de capas

"Tengo algún problema", dice el programador novato. "Quiero permitir al usuario arrastrar componentes en un contenedor de la Swing, pero cuando los componentes se arrastran, pasan por encima de unos componentes y por debajo de otros". Responde: "eso se debe a que tiene que tener en cuenta que los componentes de la Swing se presentan en capas. Debería poner lo que desea arrastrar en la capa de arrastre". "¿Cómo es eso?", pregunta el PN.

El panel de capas dentro del panel raíz contiene los componentes reales que aparecen en los applet y aplicaciones, incluyendo barras de menú y el panel de contenido. Aquí tenemos el diagrama de herencia para la clase del panel de capas, JLayeredPane:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JLayeredPane
```

JLayeredPane divide su rango de profundidad en varias capas distintas. Al situar un componente en una de estas capas es más sencillo asegurarnos de esta forma de que el componente se superpone de forma adecuada, sin tener que preocuparnos de la especificación de números para asignar profundidades específicas:

- ***DEFAULT_LAYER***. La capa estándar más baja, donde se sitúan la mayoría de los componentes.
- ***PALETTE_LAYER***. La capa de paleta está situada por encima de la capa por defecto y se utiliza para las barras de herramientas y paletas flotantes.
- ***MODAL_LAYER***. La capa utilizada por los cuadros de diálogo modales.
- ***POPUP_LAYER***. La capa emergente se visualiza sobre la capa de diálogo.
- ***DRAG_LAYER***. Cuando arrastramos un componente, asignado a la capa de arrastre, nos aseguramos de que queda posicionado sobre todos los demás componentes en el contenido.

Puede utilizar los métodos moveToFront, moveToBack y setPosition de JLayeredPane para reposicionar un componente dentro de su capa. El método setLayer también se puede utilizar para cambiar la capa actual del componente. Examinamos en el capítulo 11 JLayeredPane y podemos encontrar los campos de la clase JLayeredPane en la tabla 11.10, su constructor en la tabla 11.11 y sus métodos en la tabla 11.12.

Aquí tenemos un ejemplo que muestra cómo añadir componentes a un panel de capas. En este caso, reemplazamos el panel de contenido de un applet con un nuevo panel de capas y añadimos una etiqueta a cada etapa estándar. Para asignar la capa actual en el panel de capas, utilizamos el método setLayer y para especificar la capa que utilizamos, utilizamos las constantes definidas en la clase JLayeredPane, como JLayeredPane. PALETTE_LAYER. Una curiosidad de la Swing es que las constantes como JLayeredPane.PALETTE_LAYER son, en realidad, objetos Integer, no enteros. No obstante, los métodos como setLayer necesitan enteros, por lo que debemos usar el método intValue de la clase Integer para convertir estas constantes a valores útiles, como a continuación:

```
JLayeredPane.PALETTE_LAYER.intValue()
```

Aquí añadimos etiquetas a todas las capas descritas en la lista anterior:

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

/*
  <APPLET
    CODE = layeredpane.class
    WIDTH = 350
    HEIGHT = 280 >
  </APPLET>
*/

public class layeredpane extends JApplet
{
    JLayeredPane jlayeredpane = new JLayeredPane();

    JLabel labels[];

    public void init()
    {
        setContentPane(jlayeredpane);
        jlayeredpane.setLayout(null);

        labels = new JLabel[6];

        labels[0] = new JLabel("Capa de contenido");
        labels[0].setOpaque(true);
```

```
            labels[0].setBorder(BorderFactory.createEtchedBorder());
                                    jlayeredpane.setLayer(labels[0],
JLayeredPane.FRAME_CONTENT_LAYER.intValue());
            jlayeredpane.add(labels[0]);

            labels[1] = new JLabel("Capa predeterminada");
            labels[1].setOpaque(true);
            labels[1].setBorder(BorderFactory.createEtchedBorder());
                                    jlayeredpane.setLayer(labels[1],
JLayeredPane.DEFAULT_LAYER.intValue());
            jlayeredpane.add(labels[1]);

            labels[2] = new JLabel("Capa de paleta");
            labels[2].setOpaque(true);
            labels[2].setBorder(BorderFactory.createEtchedBorder());
                                    jlayeredpane.setLayer(labels[2],
JLayeredPane.PALETTE_LAYER.intValue());
            jlayeredpane.add(labels[2]);

            labels[3] = new JLabel("Capa modal");
            labels[3].setOpaque(true);
            labels[3].setBorder(BorderFactory.createEtchedBorder());
                                    jlayeredpane.setLayer(labels[3],
JLayeredPane.MODAL_LAYER.intValue());
            jlayeredpane.add(labels[3]);

            labels[4] = new JLabel("Capa emergente");
            labels[4].setOpaque(true);
            labels[4].setBorder(BorderFactory.createEtchedBorder());
                                    jlayeredpane.setLayer(labels[4],
JLayeredPane.POPUP_LAYER.intValue());
            jlayeredpane.add(labels[4]);

            labels[5] = new JLabel("Capa de arrastre");
            labels[5].setOpaque(true);
            labels[5].setBorder(BorderFactory.createEtchedBorder());
                                    jlayeredpane.setLayer(labels[5],
JLayeredPane.DRAG_LAYER.intValue());
            jlayeredpane.add(labels[5]);


            for(int loop_index = 0; loop_index < 6; loop_index++) {
             labels[loop_index].setBounds(40 * loop_index, 40 * loop_index,
100, 60);
            }
        }
    }
```

El resultado de este código se muestra en la figura 15.3, donde puede ver qué capas están por encima de otras. Ahora ya hemos trabajado con todas las capas estándar en un panel de capas (también podemos definir nuestras propias capas de forma numérica). Este ejemplo se encuentra en el archivo layeredpane.java del CD.

Como se muestra en la figura 15.3, puede observar que la capa de arrastre aparece por encima, lo que tiene sentido debido a que queremos que los componentes que el usuario arrastre pasen por encima de otros componentes. Para implementar el arrastre utilizando la capa de arrastre, podemos añadir el componente que queramos arrastrar a la capa de arrastre, asegurando que el gestor de distribución se haya asignado a nulo y después utilizar el método setLocation del componente para situarlo como respuesta a las acciones de ratón del usuario.

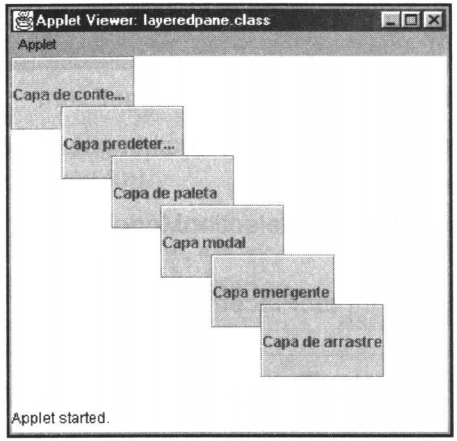


Figura 15.3. Adición de componentes a un panel de capas.

Creación de paneles de lengüetas

"Vaya", dice el programador novato. "Mi programa tiene tantas configuraciones ahora que el cuadro de diálogo de configuración es mayor que la pantalla". "¿Cuánto mayor? ", preguntamos. "Como cuatro veces", dice el PN. "Ah", decimos, "parece que es el momento de distribuir estas opciones utilizando un panel de lengüetas".

Los paneles de lengüetas proporcionan una forma cómoda de distribuir muchos componentes en un espacio pequeño. Presenta al usuario una vista basada en las lengüetas de las carpetas de archivo y cuando el usuario hace clic sobre las diversas lengüetas, parece que se abren las distintas "carpetas" mostrando un nuevo panel completo de componentes. Los paneles de lengüetas quedan soportados en la Swing con la clase JTabbedPane y aquí tenemos el diagrama de herencia para esta clase:

```
java.lang.Object
  |
```

```
+-java.awt.Component
        |
        +-java.awt.Container
                |
                +-javax.swing.JComponent
                        |
                        +-javax.swing.JTabbedPane
```

La tabla 15.1 muestra los campos para la clase JTabbedPane, la tabla 15.2 sus constructores y la tabla 15.3 sus métodos.

Tabla 15.1. Campos de la clase JTabbedPane.

Campo	Descripción
protected ChangeEvent changeEvent	El ChangeEvent.
protected ChangeListener changeListener	El ChangeListener añadido al modelo.
protected SingleSelectionModel model	El modelo de selección predeterminado.
protected int tabPlacement	Dónde situamos las lengüetas.

Tabla 15.2. Constructores de la clase JTabbedPane.

Constructor	Descripción
JTabbedPane()	Construye un JTabbedPane vacío.
JTabbedPane(int tabPlacement)	Construye un JTabbedPane vacío con la disposición de lengüetas indicada por TOP, BOTTOM, LEFT, o RIGHT.

Tabla 15.3. Métodos de la clase JTabbedPane.

Método	Descripción
Component add(Component component)	Añade un componente.
Component add(Component component, int index)	Añade un componente en la posición de lengüeta indicada por el índice.
void add(Component component, Object constraints)	Añade un componente al panel de lengüetas.
void add(Component component, Object constraints, int index)	Añade un componente al índice de lengüetas indicado.

Método	Descripción
Component add(String title, Component component)	Añade un componente con un título de lengüetas indicado.
void addChangeListener(ChangeListener l)	Añade un ChangeListener a este panel de lengüetas.
void addTab(String title, Component component)	Añade una lengüeta representada por un título y un icono.
void addTab(String title, Icon icon, Component component)	Añade la lengüeta representada por un título y/o icono, pudiendo ser cualquiera de los dos nulos.
void addTab(String title, Icon icon, Component component, String tip)	Añade una lengüeta de ayuda representada por un título y/o icono, cualquiera de los cuales puede ser nulo.
protected ChangeListener createChangeListener()	Sobreescribe éste método para devolver una subclase de ModelListener u otra implementación de ChangeListener.
protected void fireStateChanged()	Envía un ChangeEvent a cada receptor.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext asociado con este JComponent.
Color getBackgroundAt(int index)	Obtiene el color de fondo de lengüetas en index.
Rectangle getBoundsAt(int index)	Obtiene los límites de lengüetas en index.
Component getComponentAt(int index)	Obtiene el componente en index.
Icon getDisabledIconAt(int index)	Obtiene el icono de la lengüeta deshabilitada en index.
Color getForegroundAt(int index)	Obtiene el color de primer plano de la lengüeta en index.
Icon getIconAt(int index)	Obtiene el icono de lengüetas en index.

Método	Descripción
SingleSelectionModel getModel()	Obtiene el modelo asociado con este panel de lengüetas.
Component getSelectedComponent()	Obtiene el componente seleccionado en curso para este panel de lengüetas.
int getSelectedIndex()	Obtiene el índice seleccionado actual para este panel de lengüetas.
int getTabCount()	Obtiene el número de lengüetas de este panel de lengüetas.
int getTabPlacement()	Obtiene la situación de las lengüetas para este panel de lengüetas.
int getTabRunCount()	Obtiene el número de lengüetas en ejecución actualmente en uso para visualizar lengüetas.
String getTitleAt(int index)	Obtiene el título de la lengüeta en index.
String getToolTipText(MouseEvent event)	Obtiene el texto de la ayuda de herramientas para el componente determinado por la posición del evento de ratón.
TabbedPaneUI getUI()	Obtiene el objeto IU que implementa la apariencia de este componente.
String getUIClassID()	Obtiene el nombre de la clase IU que implementa la apariencia de este componente.
int indexOfComponent(Component component)	Obtiene el índice del componente lengüeta indicado.
int indexOfTab(Icon icon)	Obtiene el índice de la primera de las lengüetas con un icono dado.
int indexOfTab(String title)	Obtiene el índice de la primera lengüeta con un título dado y devuelve -1 si no existen lengüetas con este título.

Método	Descripción
void insertTab(String title, Icon icon, Component component, String tip, int index)	Inserta un componente representado por un título e icono.
boolean isEnabledAt(int index)	Obtiene si la lengüeta en index está actualmente habilitada.
protected String paramString()	Obtiene una representación de cadena de este JTabbedPane.
void remove(Component component)	Elimina la lengüeta que corresponde al componente indicado.
void removeAll()	Elimina todas las lengüetas del panel de lengüetas.
void removeChangeListener(ChangeListener l)	Elimina un ChangeListener de este panel de lengüetas.
void removeTabAt(int index)	Elimina la lengüeta en index.
void setBackgroundAt(int index, Color background)	Asigna el color de fondo.
void setComponentAt(int index, Component component)	Asigna el componente en index a Component.
void setDisabledIconAt(int index, Icon disabledIcon)	Asigna el icono inhabilitado en index a Icono, que puede ser nulo.
void setEnabledAt(int index, boolean enabled)	Asigna si la lengüeta en index está habilitada.
void setForegroundAt(int index, Color foreground)	Asigna el color de primer plano.
void setIconAt(int index, Icon icon)	Asigna el icono.
void setModel(SingleSelectionModel model)	Asigna al modelo a utilizar con este panel de lengüetas.
void setSelectedComponent(Component c)	Asigna el componente seleccionado por este panel de lengüetas.
void setSelectedIndex(int index)	Asigna el índice seleccionado para este panel de lengüetas.

Método	Descripción
void setTabPlacement (int tabPlacement)	Asigna la posición de las lengüetas para este panel de lengüetas.
void setTitleAt(int index, String title)	Asigna el título.
void setUI(TabbedPaneUI ui)	Asigna el objeto IU que implementa la apariencia de este componente.
void updateUI()	Llamado por UIManager cuando cambia la apariencia.

Un ejemplo lo dejará más claro. En este caso, crearemos un panel de lengüetas con tres lengüetas y damos a cada una de ellas su propio objeto JPanel conteniendo una etiqueta. El usuario será capaz de pasar de una lengüeta a otra, visualizando todas las etiquetas sucesivamente. Comenzaremos creando los paneles:

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

/*
  <APPLET
    CODE = tabbedpane.class
    WIDTH = 400
    HEIGHT = 200 >
  </APPLET>
*/

public class tabbedpane extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JPanel jpanel1 = new JPanel();
        JPanel jpanel2 = new JPanel();
        JPanel jpanel3 = new JPanel();

        jpanel1.add(new JLabel("Este es el panel 1"));
        jpanel2.add(new JLabel("Este es el panel 2"));
        jpanel3.add(new JLabel("Este es el panel 3"));

...
```

Ahora, estamos listos para añadir posteriormente paneles a un panel de lengüetas, por lo que crearemos un nuevo panel y utilizamos el método

addTab para añadirle 3 lengüetas. Pasaremos a addTab los títulos de cada lengüeta, los iconos de imagen que vamos a utilizar ellas, los componentes que añadiremos a cada una (que normalmente serán objetos de la clase JPanel) y una ayuda de herramientas para cada una de las lengüetas como sigue:

```
public class tabbedpane extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JTabbedPane jtabbedpane = new JTabbedPane();

        JPanel jpanel1 = new JPanel();
        JPanel jpanel2 = new JPanel();
        JPanel jpanel3 = new JPanel();

        jpanel1.add(new JLabel("Este es el panel 1"));
        jpanel2.add(new JLabel("Este es el panel 2"));
        jpanel3.add(new JLabel("Este es el panel 3"));

        jtabbedpane.addTab("Lengüeta 1",
                    new ImageIcon("tab.jpg"),
                    jpanel1, "Esta es la lengüeta 1");

        jtabbedpane.addTab("Lengüeta 2",
                    new ImageIcon("tab.jpg"),
                    jpanel2, "Esta es la lengüeta 2");

        jtabbedpane.addTab("Lengüeta tres",
                    new ImageIcon("tab.jpg"),
                    jpanel3, "Esta es la lengüeta 3");

        contentPane.setLayout(new BorderLayout());
        contentPane.add(jtabbedpane);
    }
}
```

La figura 15.4 muestra el resultado. Como se ve, cada lengüeta tiene su propia imagen y título, como especifica el método addTab. Si hacemos clic sobre una nueva lengüeta abrimos una nueva "carpeta" que contiene una etiqueta.

Eso es todo. Este ejemplo se encuentra en el archivo tabbedpane.java del CD.

Puede hacer mucho más con paneles de lengüetas; también puede especificar la posición de las lengüetas, por ejemplo. Consulte el siguiente tema para más detalles.

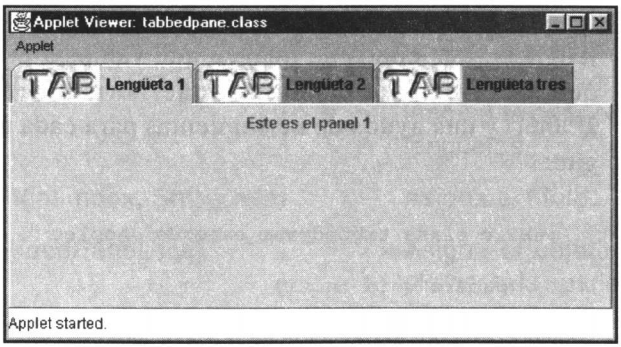


Figura 15.4. Creación de un panel de lengüetas.

Especificación de la posición de las lengüetas en los paneles de lengüetas

El gran jefe mira por encima de su hombro y dice, "No, no, todo está mal; las lengüetas debían ir al lado *izquierdo*". "Cambia de opinión continuamente", le contesta. "¿Está seguro de que quiere las etiquetas a la izquierda?" "No, no", dice el jefe, "en el lado *derecho*, como he dicho".

Podemos especificar dónde aparecen las lengüetas en un panel (arriba, abajo, izquierda o derecha) utilizando el método setTabPlacement. Aquí tenemos un ejemplo que muestra cómo funciona. En este caso, permitimos al usuario seleccionar la orientación de las lengüetas en un panel de lengüetas (izquierda, derecha, arriba o abajo) haciendo clic sobre un botón. Aquí tenemos el código:

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

/*
  <APPLET
    CODE = tabbedpaneorientation.class
    WIDTH = 500
    HEIGHT = 200 >
  </APPLET>
*/

public class tabbedpaneorientation extends JApplet implements ActionListener
{
    JTabbedPane jtabbedpane = new JTabbedPane(SwingConstants.BOTTOM);
    JButton button1, button2, button3, button4;

    public void init()
```

```
{
    Container contentPane = getContentPane();

    JPanel buttonPanel = new JPanel();

    JPanel jpanel1 = new JPanel();
    JPanel jpanel2 = new JPanel();
    JPanel jpanel3 = new JPanel();

    jtabbedpane.setTabPlacement(JTabbedPane.TOP);

    jtabbedpane.addTab("Panel 1",
                new ImageIcon("tab.jpg"),
                jpanel1, "Este es el panel 1");

    jtabbedpane.addTab("Panel 2",
                new ImageIcon("tab.jpg"),
                jpanel2, "Este es el panel 2");

    jtabbedpane.addTab("Panel 3",
                new ImageIcon("tab.jpg"),
                jpanel3, "Este es el panel 3");

    button1 = new JButton("Arriba");
    button2 = new JButton("Izquierda");
    button3 = new JButton("Derecha");
    button4 = new JButton("Abajo");

    buttonPanel.add(button1);
    buttonPanel.add(button2);
    buttonPanel.add(button3);
    buttonPanel.add(button4);

    button1.addActionListener(this);
    button2.addActionListener(this);
    button3.addActionListener(this);
    button4.addActionListener(this);

    contentPane.setLayout(new BorderLayout());

    contentPane.add(jtabbedpane, BorderLayout.CENTER);
    contentPane.add(buttonPanel, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent e)
{
    if(e.getSource() == button1) {
        jtabbedpane.setTabPlacement(JTabbedPane.TOP);
    }
    else if(e.getSource() == button2) {
        jtabbedpane.setTabPlacement(JTabbedPane.LEFT);
    }
    else if(e.getSource() == button3) {
        jtabbedpane.setTabPlacement(JTabbedPane.RIGHT);
    }
```

```
            else if(e.getSource() == button4) {
                jtabbedpane.setTabPlacement(JTabbedPane.BOTTOM);
            }
                jtabbedpane.validate();
        }
    }
```

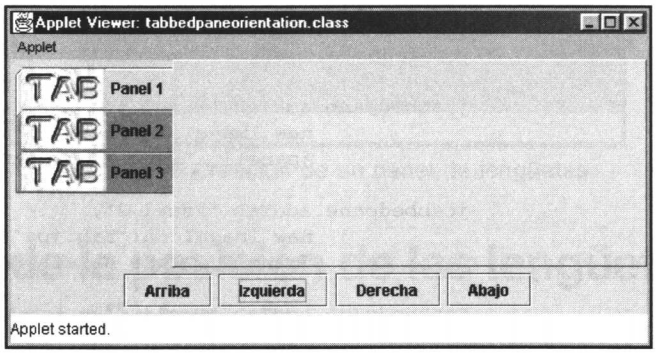


Figura 15.5. Creación de un panel de lengüetas.

La figura 15.5 muestra el resultado de este código. Cuando el usuario hace clic sobre un botón, el panel de lengüetas responde visualizando las lengüetas en la orientación correspondiente, permitiendo a los usuarios personalizar el panel a su gusto. El gran jefe estaría orgulloso. Este ejemplo se encuentra en el archivo tabbedpaneorientation.java del CD.

Uso de paneles de separación

El especialista de soporte a productos vuelve y no está feliz. "Los usuarios comentan sobre nuestro nuevo programa procesador de texto", dice el ESP. "Cuando editan un documento de más de mil páginas y tienen que hacer referencia a otra parte del documento, puede requerir mucho tiempo desplazarse hacia arriba y hacia abajo". "¿Quién está editando documentos de más de mil páginas?", preguntamos. "¿Qué tal si añadimos un panel de separación al procesador de texto?", pregunta el ESP.

Podemos utilizar paneles de separación para visualizar dos componentes colindantes, que pueden incluir dos vistas del mismo modelo. Utilizando paneles de separación, el usuario puede gestionar dos componentes, arrastrar el separador entre los componentes para mostrar más o menos de cada uno según sea necesario. Los paneles de separación están soportados por la clase JSplitPane y aquí tenemos el árbol de herencia para esta clase:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JSplitPane
```

La tabla 15.4 muestra los campos de la clase JSplitPane, la tabla 15.5 muestra sus constructores y la tabla 15.6 sus métodos.

Tabla 15.4. Campos de la clase JSplitPane.

Campo	Descripción
static String BOTTOM	Utilizado para añadir un componente en el pie de panel.
static String CONTINUOUS_LAYOUT_PROPERTY	El nombre de la propiedad ligada para continuousLayout.
protected boolean continuousLayout	Devuelve true si las vistas se dibujan continuamente mientras se cambia de tamaño.
static String DIVIDER	Utilizado para añadir un componente que representará al divisor.
static String DIVIDER_SIZE_PROPERTY	Nombre de la propia ligada para el borde.
protected int dividerSize	El tamaño del divisor.
static int HORIZONTAL_SPLIT	Un divisor horizontal indica que el componente se divide a lo largo del eje x.
static String LAST_DIVIDER_LOCATION_PROPERTY	La propiedad ligada para lastLocation.
protected int lastDividerLocation	La localización previa de paneles de separación.
static String LEFT	Utilizado para añadir un componente a la izquierda del otro componente.
protected Component leftComponent	El componente que hay a la izquierda o arriba.
static String ONE_TOUCH_EXPANDABLE_PROPERTY	La propiedad ligada para oneTouchExpandable.

Campo	Descripción
protected boolean oneTouchExpandable	Devuelve cierto si el panel de separación es expandible con una pulsación.
protected int orientation	Indica cómo se separan las vistas.
static String ORIENTATION_PROPERTY	El nombre de la propiedad ligada para la orientación (horizontal o vertical).
static String RIGHT	Utilizado para añadir un componente a la derecha del componente.
protected Component rightComponent	El componente a la derecha o al pie.
static String TOP	Utilizado para añadir un componente sobre otro componente.
static int VERTICAL_SPLIT	Una separación vertical indica que los componentes se dividen a lo largo del eje y.

Tabla 15.5. Constructores de la clase JSplitPane.

Constructor	Descripción
JSplitPane()	Construye un JSplitPane con los botones para los componentes.
JSplitPane(int newOrientation)	Construye un JSplitPane configurado con la distribución no continua indicada.
JSplitPane(int newOrientation, boolean newContinuousLayout)	Construye un JSplitPane nuevo con la orientación y estilo de dibujo indicados.
JSplitPane(int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)	Construye un JSplitPane con la orientación y estilo de dibujo indicados y con los componentes indicados.
JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)	Construye un JSplitPane con la orientación y los componentes especificados, que no realiza dibujo continuo.

Tabla 15.6. Métodos de la clase JSplitPane.

Método	Descripción
protected void addImpl(Component comp, Object constraints, int index)	Añade *comp* con las limitaciones dadas.
AccessibleContext getAccessible-Context()	Obtiene el AccessibleContext.
Component getBottomComponent()	Obtiene el componente por debajo a la derecha del divisor.
int getDividerLocation()	Obtiene la localización del divisor a partir de una implementación de la apariencia.
int getDividerSize()	Obtiene el tamaño del divisor.
int getLastDividerLocation()	Obtiene la última posición del divisor.
Component getLeftComponent()	Obtiene el componente a la izquierda o por encima del divisor.
int getMaximumDividerLocation()	Obtiene la posición máxima del divisor a partir de la implementación de la apariencia.
int getMinimumDividerLocation()	Obtiene la posición mínima del divisor a partir de implementación de la apariencia.
int getOrientation()	Obtiene la orientación.
Component getRightComponent()	Obtiene el componente a la derecha o por debajo del divisor.
Component getTopComponent()	Obtiene el componente a la izquierda por encima del divisor.
SplitPaneUI getUI()	Obtiene el SplitPaneUI que proporciona la apariencia actual.
String getUIClassID()	Obtiene el nombre de la clase de la apariencia que renderiza este componente.
boolean isContinuousLayout()	Devuelve cierto si los componentes fijos se dibujan continuamente y se distribuyen durante la intervención del usuario.

Método	Descripción
boolean isOneTouchExpandable()	Devuelve cierto si el panel proporciona un elemento IU para reducir/expandir el divisor.
boolean isValidateRoot()	Devuelve cierto si la raíz es válida.
protected void paintChildren(Graphics g)	Pinta todos componentes.
protected String paramString()	Devuelve una representación de cadena de este JSplitPane.
void remove(Component component)	Elimina el componente hijo del panel.
void remove(int index)	Elimina el componente en el índice indicado.
void removeAll()	Elimina todo los componentes hijos del receptor.
void resetToPreferredSizes()	Reposiciona el JSplitPane basándose en el tamaño preferido de los hijos.
void setBottomComponent(Component comp)	Asigna el componente arriba o a la derecha del visor.
void setContinuousLayout(boolean newContinuousLayout)	Determina si los componentes fijos se dibujarán continuamente durante las acciones de usuario.
void setDividerLocation(double proportionalLocation)	Asigna la localización del divisor con un porcentaje del tamaño del JSplitPane.
void setDividerLocation(int location)	Asigna la localización del divisor.
void setDividerSize(int newSize)	Asigna el tamaño de divisor.
void setLastDividerLocation(int newLastLocation)	Asigna la última localización del divisor que estaba en newLastLocation.
void setLeftComponent(Component comp)	Asigna el componente a la izquierda o en la parte superior del divisor.
void setOneTouchExpandable(boolean newValue)	Determina si JSplitPane proporciona un elemento IU del divisor para expandir/reducir rápidamente el divisor.

Método	Descripción
void setOrientation(int orientation)	Asigna la orientación o la división del separador.
void setRightComponent(Component comp)	Asigna el componente a la derecha o por debajo del divisor.
void setTopComponent(Component comp)	Asigna el componente por encima o a la izquierda del separador.
void setUI(SplitPaneUI ui)	Asigna el objeto apariencia de dibujo de componentes.
void updateUI()	Llamado por UIManager cuando cambia la apariencia.

Crearemos un ejemplo con la clase JSplitPane. En este caso, añadimos los campos de texto a un panel divisor. Hacer esto es bastante fácil; basta con crear los campos de texto y un panel de separación, como hacemos a continuación:

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

/*
  <APPLET
    CODE = splitpane.class
    WIDTH = 600
    HEIGHT = 200 >
  </APPLET>
*/

public class splitpane extends JApplet implements ActionListener
{
    JButton jbutton1, jbutton2, jbutton3;
    JTextField text1 = new JTextField("Texto 1");
    JTextField text2 = new JTextField("Texto 2");
    JSplitPane jsplitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        text1, text2);
  ...
}
```



Truco: Si no añade explícitamente componentes a un panel de separación, éste mostrará dos botones.

Ahora añadimos el nuevo panel de separación al panel de contenidos del applet, como sigue:

```
public class splitpane extends JApplet implements ActionListener
{
    JButton jbutton1, jbutton2, jbutton3;
    JTextField text1 = new JTextField("Texto 1");
    JTextField text2 = new JTextField("Texto 2");
    JSplitPane jsplitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        text1, text2);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.add(jsplitpane, BorderLayout.CENTER);
    }
}
```

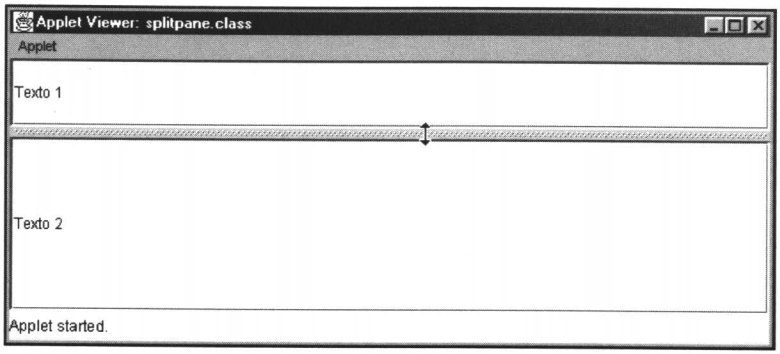


Figura 15.6. Creación de un panel de separación.

Eso es todo. El resultado de este código se muestra en la figura 15.6. A partir de ahora, el usuario puede ajustar el divisor entre los campos de texto mostrando una cantidad mayor o menor de cada uno según sea necesario. El panel de separación gestionará la vista de cada campo de texto. Este ejemplo se encuentra en el archivo splitpane.java del CD.

Paneles de separación expandibles con un clic

"Examina el nuevo panel de separación que he puesto a mi programa", dice el programador novato. "Está bien", reconocemos, "pero ¿por qué no lo hace expandirse con un clic?". El PN replica, "¿Cómo?".

Hacer que un panel de separación sea expandible con una pulsación de ratón añade pequeños botones de flecha a su separador, los cuales, cuando son pulsados, expedirán uno u otro de los paneles para que ocupe el panel de separación completo. Este es un mecanismo útil si queremos maximizar un

componente en un panel de separación, debido a que los paneles de separación no permiten la reducción de los componentes por debajo de un cierto mínimo. Podemos hacer que un panel de separación sea expandible con una pulsación con el método setOneTouchExpandable del panel de separación. Para mostrarle cómo funciona, añadiremos un botón al panel de separación del ejemplo del tema previo que, cuando se pulse, hará que el panel de separación se expanda con una pulsación. Para añadir ese botón, crearemos primero un nuevo panel y lo añadiremos al distribuidor del applet, como sigue:

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

/*
<APPLET
    CODE = splitpane.class
    WIDTH = 600
    HEIGHT = 200 >
</APPLET>
*/

public class splitpane extends JApplet implements ActionListener
{
    JButton jbutton1;
    JTextField text1 = new JTextField("Texto 1");
    JTextField text2 = new JTextField("Texto 2");
    JSplitPane jsplitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        text1, text2);

    public void init()
    {
        Container contentPane = getContentPane();
        JPanel jpanel = new JPanel();

        jbutton1 = new JButton("Expandible con un clic");
        jbutton1.addActionListener(this);
        jpanel.add(jbutton1);

        contentPane.add(jsplitpane, BorderLayout.CENTER);
        contentPane.add(jpanel, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            jsplitpane.setOneTouchExpandable(true);
        }
    }
}
```

La figura 15.7 muestra el resultado de este código. Cuando el usuario hace clic sobre el botón, aparecen dos flechas en el divisor del panel de separación,

como se muestra en la figura. Ahora, el panel de separación se expande con la pulsación; por ejemplo, cuando el usuario hace clic sobre la flecha superior, el campo de texto del pie se expande para llenar el panel de separación (exceptuando al divisor, que todavía es visible para que el usuario pueda hacer clic sobre la flecha inferior y restaurar la apariencia original).

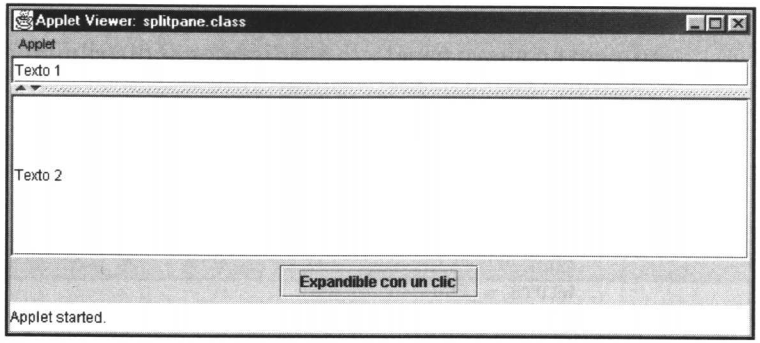


Figura 15.7. Hacer un panel de separación expandible con una pulsación.

Configuración de la orientación del panel de separación

"No, no, no", dice impaciente el gran jefe, "no queremos ningún divisor horizontal en los paneles de separación. Queremos divisores *verticales*". "Bien", decimos, "podemos hacerlo. ¿Cuándo lo quiere?" "¿Está hecho ya?", pregunta el gran jefe.

Podemos configurar la orientación del divisor en un panel de separación pasando al método setOrientation la constante JSplitPane.HORIZONTAL_SPLIT o JSplitPane.VERTICAL_SPLIT. Añadiremos esta capacidad al ejemplo del panel de separación del tema previo con un nuevo botón que permite al usuario especificar un divisor horizontal, como aparece a continuación:

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

/*
   <APPLET
    CODE = splitpane.class
    WIDTH = 600
    HEIGHT = 200 >
   </APPLET>
*/

public class splitpane extends JApplet implements ActionListener
```

```
{
    JButton jbutton1, jbutton2;
    JTextField text1 = new JTextField("Texto 1");
    JTextField text2 = new JTextField("Texto 2");
    JSplitPane jsplitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        text1, text2);

    public void init()
    {
        Container contentPane = getContentPane();
        JPanel jpanel = new JPanel();

        jbutton1 = new JButton("Expandible con un clic");
        jbutton1.addActionListener(this);
        jpanel.add(jbutton1);

        jbutton2 = new JButton("Divisible horizontalmente");
        jbutton2.addActionListener(this);
        jpanel.add(jbutton2);

        contentPane.add(jsplitpane, BorderLayout.CENTER);
        contentPane.add(jpanel, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            jsplitpane.setOneTouchExpandable(true);
        }
        if(e.getSource() == jbutton2) {
            jsplitpane.setOrientation(JSplitPane.HORIZONTAL_SPLIT);
        }
    }
}
```

La figura 15.8 muestra el resultado de este código y el nuevo divisor orientado horizontalmente.

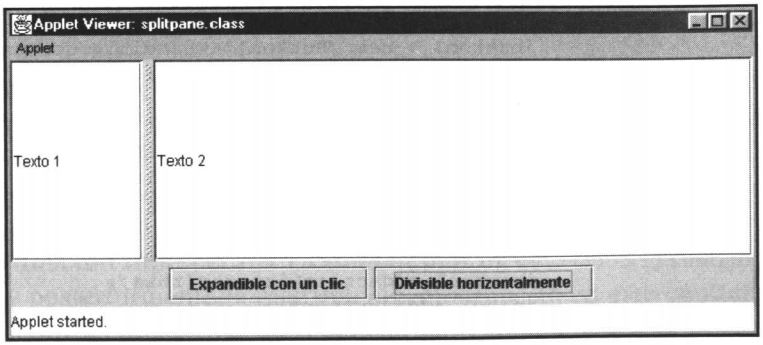


Figura 15.8. Configuración de la orientación de un panel de separación.

Configuración del tamaño del divisor de un panel de separación

Puede ajustar el tamaño del divisor que aparece en un panel de separación utilizando el método setDividerSize y pasándole la nueva anchura del divisor en puntos. Aquí tenemos un ejemplo en el que añadimos un botón al caso anterior. Cuando el usuario hace clic sobre este botón, obtiene la anchura actual del divisor con el método getDividerSize, le suma 10 puntos y utiliza el resultado como nueva anchura de divisor. Aquí tenemos el código:

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

/*
  <APPLET
    CODE = splitpane.class
    WIDTH = 600
    HEIGHT = 200 >
  </APPLET>
*/

public class splitpane extends JApplet implements ActionListener
{
    JButton jbutton1, jbutton2, jbutton3;
    JTextField text1 = new JTextField("Texto 1");
    JTextField text2 = new JTextField("Texto 2");
    JSplitPane jsplitpane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        text1, text2);

    public void init()
    {
        Container contentPane = getContentPane();
        JPanel jpanel = new JPanel();

        jbutton1 = new JButton("Expandible con un clic");
        jbutton1.addActionListener(this);
        jpanel.add(jbutton1);

        jbutton2 = new JButton("Divisible horizontalmente");
        jbutton2.addActionListener(this);
        jpanel.add(jbutton2);

        jbutton3 = new JButton("Incrementar el tamaño del divisor");
        jbutton3.addActionListener(this);
        jpanel.add(jbutton3);

        contentPane.add(jsplitpane, BorderLayout.CENTER);
        contentPane.add(jpanel, BorderLayout.SOUTH);
```

```
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            jsplitpane.setOneTouchExpandable(true);
        }
        if(e.getSource() == jbutton2) {
            jsplitpane.setOrientation(JSplitPane.HORIZONTAL_SPLIT);
        }
        if(e.getSource() == jbutton3) {
          jsplitpane.setDividerSize(jsplitpane.getDividerSize() + 10);
        }
    }
}
```

La figura 15.9 muestra el resultado. Cuando el usuario hace clic sobre el botón para incrementar el tamaño del divisor, éste aumenta su tamaño cada vez en 10 puntos.

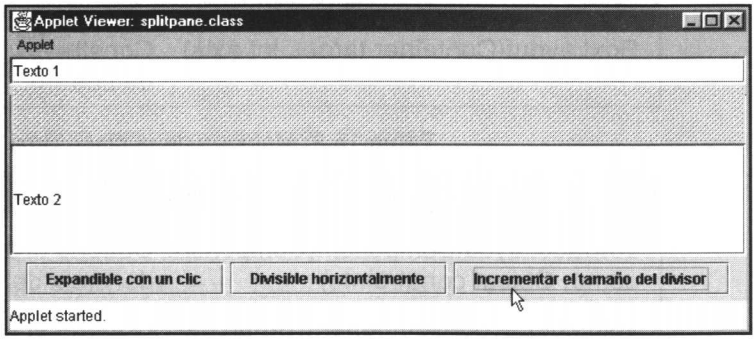


Figura 15.9. Configuración del tamaño de un divisor de un panel horizontal.

Uso del gestor de distribución de cuadro

"Tengo muchas herramientas de dibujo en mi programa", dice el programador novato, "y quiero distribuirlas verticalmente en una columna alta. ¿Alguna sugerencia?" "Claro", decimos, "utilice un distribuidor de cuadro". "Bien", dice el PN, pero "¿cómo?"

La Swing introduce dos nuevas distribuciones: la distribución del cuadro y la distribución superpuesta. Examinaremos en este tema la distribución de cuadro y posteriormente la distribución superpuesta en este capítulo. Puede utilizar distribuciones de cuadro para situar componentes a lo largo del eje x en una fila o a lo largo del eje y en una columna. Aquí tiene el árbol de herencia para la clase BoxLayout, que soporta la distribución de cuadro:

```
java.lang.Object
  |
  +--javax.swing.BoxLayout
```

La tabla 15.7 muestra los campos de la clase BoxLayout, la tabla 15.8 muestra su constructor y la tabla 15.9 sus métodos.

Tabla 15.7. Campos de la clase BoxLayout.

Campo	Descripción
static int X_AXIS	Indica que los componentes deberían distribuirse de izquierda a derecha.
static int Y_AXIS	Indica que los componentes deberían distribuirse de arriba a abajo.

Tabla 15.8. El constructor de la clase BoxLayout.

Constructor	Descripción
BoxLayout(Container target, int axis)	Construye un gestor de distribución de cuadro.

Tabla 15.9. Métodos de la clase BoxLayout.

Método	Descripción
voidaddLayoutComponent(Component comp, Object constraints)	Obtiene el alineamiento a lo largo del eje x del contenedor.
void addLayoutComponent(String name, Component comp)	Obtiene el alineamiento a lo largo del eje y del contenedor.
float getLayoutAlignmentX(Container target)	Indica el cambio de un hijo en su información de distribución.
float getLayoutAlignmentY(Container target)	Llamado por el AWT cuando el contenedor indicado necesita una nueva distribución.
Dimension maximumLayoutSize(Container target)	Obtiene las dimensiones máximas que puede utilizar el contenedor destino para distribuir los componentes que contiene.
Dimension minimumLayoutSize(Container target)	Obtiene las dimensiones mínimas necesitadas para distribuir los componentes del contenedor destino indicado.

Método	Descripción
Dimension preferredLayoutSize(Container target)	Obtiene las dimensiones preferidas para esta distribución, dados los componentes en el contenedor destino indicado.

Aquí tenemos un ejemplo en el que distribuimos tres paneles usando una distribución de cuadro (dos verticalmente y uno horizontalmente) y añadimos cuatro campos de texto a cada distribución. Esto supone simplemente utilizar un nuevo gestor de distribución de cuadro en cada panel, como sigue:

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

/*
  <APPLET
    CODE = boxlayout.class
    WIDTH = 250
    HEIGHT = 200 >
  </APPLET>
*/

public class boxlayout extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();

        JPanel jpanel1, jpanel2, jpanel3;

        jpanel1 = new JPanel();
        jpanel2 = new JPanel();
        jpanel3 = new JPanel();

        jpanel1.setLayout(new BoxLayout(jpanel1, BoxLayout.Y_AXIS));
        jpanel2.setLayout(new BoxLayout(jpanel2, BoxLayout.X_AXIS));
        jpanel3.setLayout(new BoxLayout(jpanel3, BoxLayout.Y_AXIS));

        contentPane.setLayout(new FlowLayout());

        jpanel1.add(new JTextField("Texto 1"));
        jpanel1.add(new JTextField("Texto 2"));
        jpanel1.add(new JTextField("Texto 3"));
        jpanel1.add(new JTextField("Texto 4"));

        jpanel2.add(new JTextField("Texto 1"));
        jpanel2.add(new JTextField("Texto 2"));
        jpanel2.add(new JTextField("Texto 3"));
        jpanel2.add(new JTextField("Texto 4"));

        jpanel3.add(new JTextField("Texto 1"));
```

```
        jpanel3.add(new JTextField("Texto 2"));
        jpanel3.add(new JTextField("Texto 3"));
        jpanel3.add(new JTextField("Texto 4"));

        contentPane.add(jpanel1);
        contentPane.add(jpanel2);
        contentPane.add(jpanel3);
    }
}
```

La figura 15.10 muestra el resultado, donde puede ver las tres distribuciones de cuadro. Este ejemplo fue relativamente fácil de crear y ejecutar. Aparece en el archivo boxlayout.java del CD.

No obstante, existe una mayor potencia disponible en la distribución de cuadro; examinaremos la clase Box en el siguiente tema.

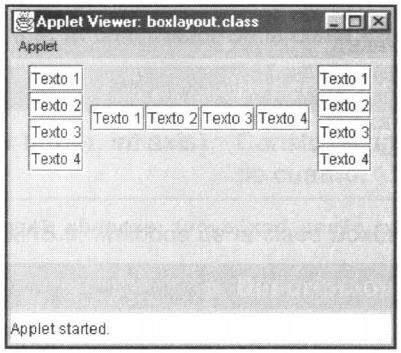


Figura 15.10. Creación de distribuciones de cuadro.

Uso de la clase Box

"Las distribuciones de cuadro están bien", dice el programador novato, "pero necesitó más control. Por ejemplo, quiero espaciar los controles en una fila, pero no puedo hacerlo con una distribución de cuadro". "Seguro que puede", contesta, "si utiliza la clase Box". "¿Cómo es eso?", pregunta el PN.

La clase Box utiliza una distribución de cuadro que añade métodos que nos permiten trabajar con estas construcciones para extender esa distribución:

- *Glue*. Expande hasta completar la región vacía entre componentes. Se puede utilizar *glue* para espaciar los componentes de forma uniforme.
- *Struts*. Son las distancias horizontales o verticales que especificamos y añadimos a la distribución para espaciar los componentes como desee-

mos. Los *struts* están fijos en una dimensión y rellenan la otra dimensión según sea necesario.

- *Áreas rígidas*. Son áreas rectangulares que no cambian de tamaño por sí mismas.

Aquí tenemos el diagrama de herencia para la clase Box:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.Box
```

Encontrará los campos de la clase Box en la tabla 15.10, sus constructores en la tabla 15.11 y sus métodos en la tabla 15.12.

Tabla 15.10. El campo de la clase Box.

Campo	Descripción
protected AccessibleContext accessibleContext	El AccessibleContext.

Tabla 15.11. El constructor de la clase Box.

Constructor	Descripción
Box(int axis)	Construye un cuadro.

Tabla 15.12. Métodos de la clase Box.

Método	Descripción
static Component createGlue()	Construye un componente pegado.
static Box createHorizontalBox()	Construye un cuadro que visualiza sus componentes de izquierda a derecha.
static Component createHorizontalGlue()	Construye un componente pegado horizontal.
static Component createHorizontalStrut(int width)	Construye una dura liga invisible de anchura fija.
static Component createRigidArea (Dimension d)	Construye un componente invisible que siempre tiene el tamaño indicado.

Método	Descripción
static Box createVerticalBox()	Construye un cuadro que visualiza sus componentes de arriba a abajo.
static Component createVerticalGlue()	Construye un componente pegado vertical.
static Component createVertical-Strut(int height)	Construye una ligadura invisible de altura fija.
AccessibleContext getAccessible-Context()	Obtiene el AccessibleContext.
void setLayout(LayoutManager l)	Genera un AWTError (un objeto Box únicamente puede utilizar un BoxLayout).

Aquí tenemos un ejemplo de uso de la clase Box. En este caso, utilizamos los métodos createGlue, createRigidArea, createHorizontalStrut y createVerticalStrut de la clase Box para crear áreas rígidas, pegadas, e irregulares. Estos componente se pueden añadir a las distribuciones de cuadro, para crear seis paneles, cada uno con una distribución de cuadro y añadir áreas de texto a los paneles utilizando áreas pegadas, irregulares y rígidas para mostrar cómo funciona todo esto. Aquí tenemos el código:

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

/*
  <APPLET
    CODE = box.class
    WIDTH = 450
    HEIGHT = 400 >
  </APPLET>
*/

public class box extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JPanel jpanel1 = new JPanel();
                jpanel1.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"Pegada"));
        jpanel1.setLayout(new BoxLayout(jpanel1, BoxLayout.X_AXIS));
        jpanel1.add(Box.createGlue());
        jpanel1.add(new JTextField("Texto 1"));
```

```
            jpanel1.add(Box.createGlue());
            jpanel1.add(new JTextField("Texto 2"));
            jpanel1.add(Box.createGlue());
            jpanel1.add(new JTextField("Texto 3"));
            jpanel1.add(Box.createGlue());
            contentPane.add(jpanel1);

            JPanel jpanel2 = new JPanel();
                    jpanel2.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"Ajustada"));
            jpanel2.setLayout(new BoxLayout(jpanel2, BoxLayout.X_AXIS));
            jpanel2.add(new JTextField("Texto 1"));
            jpanel2.add(Box.createHorizontalStrut(20));
            jpanel2.add(new JTextField("Texto 2"));
            jpanel2.add(Box.createHorizontalStrut(20));
            jpanel2.add(new JTextField("Texto 3"));
            contentPane.add(jpanel2);

            JPanel jpanel3 = new JPanel();
                    jpanel3.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(), "Rígida"));
            jpanel3.setLayout(new BoxLayout(jpanel3, BoxLayout.X_AXIS));
            jpanel3.add(Box.createRigidArea(new Dimension(10, 40)));
            jpanel3.add(new JTextField("Texto 1"));
            jpanel3.add(Box.createRigidArea(new Dimension(10, 40)));
            jpanel3.add(new JTextField("Texto 2"));
            jpanel3.add(Box.createRigidArea(new Dimension(10, 40)));
            jpanel3.add(new JTextField("Texto 3"));
            contentPane.add(jpanel3);

            JPanel jpanel4 = new JPanel();
                    jpanel4.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(), "Pegada"));
            jpanel4.setLayout(new BoxLayout(jpanel4, BoxLayout.Y_AXIS));
            jpanel4.add(Box.createGlue());
            jpanel4.add(new JTextField("Texto 1"));
            jpanel4.add(Box.createGlue());
            jpanel4.add(new JTextField("Texto 2"));
            jpanel4.add(Box.createGlue());
            jpanel4.add(new JTextField("Texto 3"));
            jpanel4.add(Box.createGlue());
            contentPane.add(jpanel4);

            JPanel jpanel5 = new JPanel();
                    jpanel5.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(), "Pegada"));
            jpanel5.setLayout(new BoxLayout(jpanel5, BoxLayout.Y_AXIS));
            jpanel5.add(new JTextField("Texto 1"));
            jpanel5.add(Box.createVerticalStrut(30));
            jpanel5.add(new JTextField("Texto 2"));
            jpanel5.add(Box.createVerticalStrut(30));
            jpanel5.add(new JTextField("Texto 3"));
            contentPane.add(jpanel5);

            JPanel jpanel6 = new JPanel();
```

```
                    jpanel6.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(), "Rígida"));
            jpanel6.setLayout(new BoxLayout(jpanel6, BoxLayout.Y_AXIS));
            jpanel6.add(Box.createRigidArea(new Dimension(40, 60)));
            jpanel6.add(new JTextField("Texto 1"));
            jpanel6.add(Box.createRigidArea(new Dimension(40, 60)));
            jpanel6.add(new JTextField("Texto 2"));
            jpanel6.add(Box.createRigidArea(new Dimension(40, 60)));
            jpanel6.add(new JTextField("Texto 3"));
            contentPane.add(jpanel6);
        }
    }
```

La figura 15.11 muestra el resultado de este código. Como puede ver, el uso de la clase Box es una forma sencilla de extender las distribuciones de cuadro y nos permite personalizar el espacio entre los componentes. Este ejemplo se encuentra en el archivo box.java del CD.

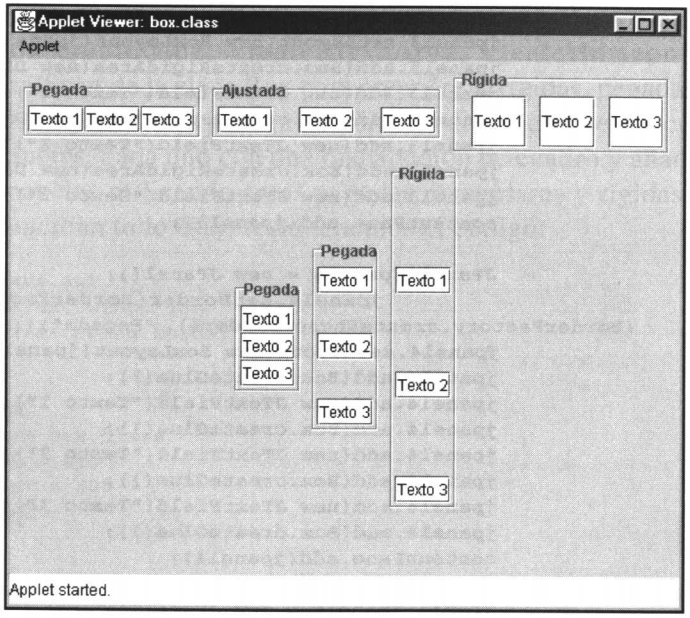


Figura 15.11. Uso de la clase Box.

Uso del gestor de distribución de superposición

"Mi programa está bastante desordenado", dice el programador novato. "¿Existe una forma sencilla para superponer los controles?" "¿Está seguro de que lo quiere hacer así?", preguntamos. "Seguro", dice el PN. "Bien", deci-

mos", "puede eliminar el gestor de distribución y localizar los controles mismos, o puede utilizar el gestor de superposición". "¿Existe un gestor de superposición?", pregunta el PN. "¡Dígame más!".

Puede utilizar el gestor de superposición para superponer componentes en la Swing. El gestor de distribución está soportado por la clase OverlayLayout. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +--javax.swing.OverlayLayout
```

La tabla 15.13 muestra el constructor para esta clase y la tabla 15.14 muestra sus métodos.

Tabla 15.13. El constructor de la clase OverlayLayout.

Constructor	Descripción
OverlayLayout(Container target)	Construye un gestor de distribución de superposición.

Tabla 15.14. Métodos de la clase OverlayLayout.

Método	Descripción
void addLayoutComponent(Component comp, Object constraints)	Añade el componente indicado a la distribución.
void addLayoutComponent(String name, Component comp)	Añade el componente indicado a la distribución.
float getLayoutAlignmentX(Container target)	Obtiene la alineación a lo largo del eje x para el contenedor.
float getLayoutAlignmentY(Container target)	Obtiene la alineación a lo largo del eje y para el contenedor.
void invalidateLayout(Container target)	Indica que un hijo ha cambiado su información relativa a la distribución, lo que produce que cualquier operación de caché se vuelque.
void layoutContainer(Container target)	Llama al AWT cuando el contenedor indicado necesita una nueva distribución.
Dimension maximumLayoutSize (Container target)	Obtiene las dimensiones máximas necesarias para distribuir los componentes.

Método	Descripción
Dimension minimumLayoutSize (Container target)	Obtiene las dimensiones mínimas necesarias para distribuir los componentes.
Dimension preferredLayoutSize (Container target)	Obtiene las dimensiones preferentes para esta distribución dados sus componentes.
void removeLayoutComponent (Component comp)	Elimina el componente indicado a partir de su distribución.

Aquí vemos cómo utilizar la distribución de superposición: En esta clase de distribución, añadimos componentes a un contenedor para que sus puntos de alineamiento estén en la misma posición. Cada uno de estos componentes también tiene unos atributos de alineamiento entre 0.0 y 1.0, que podemos asignar con los métodos setAlignmentX y setAlignmentY (el valor predeterminado es 0.5). Éstos especifican dónde está el punto de alineamiento en cada dimensión. Por ejemplo, un atributo de alineamiento de 0.5 hace referencia al centro de un componente en la dirección x.

Una vez asignados el tamaño mínimo, máximo y preferido de los componentes, el gestor de distribución de superposición intenta cambiar el tamaño de los componentes para que sus puntos de alineamiento se superpongan y procura mantener esa superposición. Si es posible, incluso aunque el contenedor cambie de tamaño.

Vamos a examinar un ejemplo para que sea más claro. En este caso, superponemos los campos de texto, especificando sus tamaños mínimo, máximo y preferido y asignando sus puntos de alineamiento para que un componente aparezca en la esquina superior izquierda y el otro en la inferior derecha. Aquí tenemos el código:

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

/*
  <APPLET
    CODE = overlay.class
    WIDTH = 200
    HEIGHT = 200 >
  </APPLET>
*/

public class overlay extends JApplet
{
    public void init()
    {
```

```
        Container contentPane = getContentPane();

        JPanel jpanel = new JPanel();
        jpanel.setLayout(new OverlayLayout(jpanel));
        jpanel.setBackground(Color.white);
                jpanel.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(), "Superpuesta"));
        JTextField text1 = new JTextField("Texto 1");
        JTextField text2 = new JTextField("Texto 2");
        text1.setMinimumSize(new Dimension(30, 30));
        text2.setMinimumSize(new Dimension(30, 30));
        text1.setPreferredSize(new Dimension(100, 100));
        text2.setPreferredSize(new Dimension(100, 100));
        text1.setMaximumSize(new Dimension(120, 120));
        text2.setMaximumSize(new Dimension(120, 120));

        text1.setAlignmentX(0.2f);
        text1.setAlignmentY(0.2f);

        text2.setAlignmentX(0.8f);
        text2.setAlignmentY(0.8f);

        jpanel.add(text1);
        jpanel.add(text2);

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jpanel);
    }
}
```

La figura 15.12 muestra el resultado, donde puede ver los dos campos de texto superpuestos. Este ejemplo es un éxito y lo encontrará en el archivo overlay.java del CD.

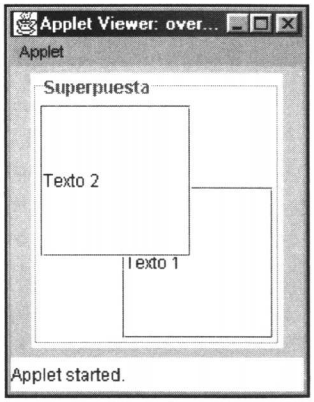


Figura 15.12. Uso de un gestor de distribución de superposición.

16 Swing: menús y barras de herramientas

En este capítulo, trataremos dos de los más importantes componentes de la Sweing: menús y barras de herramientas. Ambos componentes son familiares para todos los usuarios IU y veremos cómo funcionan en la Swing. Examinaremos primero una breve descripción de lo que la Swing ofrece.

Menús

Los menús de la Swing proporcionan algunas mejoras sustanciales sobre los menús AWT, como la habilidad para visualizar imágenes, asignar la apariencia de un elemento de menú y, especialmente, la habilidad para visualizar menús en ventanas de applet. Los menús de Swing quedan soportados por las clases JMenuBar, JMenu y JMenuItem, que soportan barras de menú, menús y elementos de menús, respectivamente. Como en la AWT, los menús de la Swing se crean utilizando botones entre bastidores, para que se puedan usar receptores de acción con ellos.

En este capítulo, examinaremos lo que ofrecen los menús de la Swing, incluyendo la creación básica de menús, adición de imágenes a menús, creación de menús de casillas de verificación y de botones de activación y

submenús, adición de controles, botones de menús, creación de mnemónicos, aceleradores de menús y mucho más. También examinaremos la creación de menús emergentes. Los menús de la Swing son más complejos y potentes que sus equivalentes en la AWT, como veremos a continuación.

Barras de herramientas

Las barras de herramientas son controles IU populares, y son nuevos en la Swing. Como los menús, las barras de herramientas se crean utilizando botones en la Swing. Las barras de herramientas proporcionan una barra de botones que se pueden pulsar como los elementos de menús. De hecho, las barras de herramientas y los menús están íntimamente relacionados; habitualmente se añadirá un botón a las barras de herramientas que representen elementos comunes de menús para evitar al usuario la molestia de abrir un menú.

Examinaremos las barras de herramientas de la Swing, que están soportadas por la clase JToolBar, incluyendo la adición de imágenes a botones de barras de herramientas, permitir al usuario alinear la barra de herramientas en cualquier borde de una ventana y la adición a las barras de herramientas de otros controles como cuadros combinados.

Esto es suficiente para resumir el capítulo. Como puede ver, habrá mucho más.

Crear una barra de menús

El programador novato aparece y dice, "El gran jefe dice que necesito añadir un sistema de menús a mi programa. ¿Cómo diablos puedo hacerlo?" "Con tres componentes", decimos, "barras de menú, menús y elementos de menús. Pida un café y comenzaremos con las barras de menús". "Bien", dice el PN.

Para añadir un sistema de menús a un programa Swing, tendremos que crear primero una barra de menús, utilizando la clase JMenuBar. Puede añadir una barra de menús al applet o marcos de ventana con el método setJMenuBar. Aquí tiene el diagrama de herencia para la clase JMenuBar:

```
java.lang.Object
  |
  +--java.awt.Component
        |
```

```
+-java.awt.Container
      |
      +-javax.swing.JComponent
            |
            +-javax.swing.JMenuBar
```

La tabla 16.1 muestra los constructores de JMenuBar y la tabla 16.2 sus métodos.

Tabla 16.1. El constructor de la clase JMenuBar.

Constructor	Descripción
JMenuBar()	Crea una nueva barra de menú.

Tabla 16.2. Métodos de la clase JMenuBar.

Método	Descripción
JMenu add(JMenu c)	Añade el menú indicado al final de la barra de menú.
void addNotify()	Sobrescribe JComponent. addNotify() para registrar esta barra de menú.
AccessibleContext getAccessible Context()	Obtiene el AccessibleContext asociado con este JComponent.
Component getComponent()	Implementado para ser un Component.
Component getComponent-AtIndex(int i)	Obtiene el componente en el índice indicado.
int getComponentIndex (Component c)	Obtiene el índice del componente indicado.
JMenu getHelpMenu()	Obtiene el menú de ayuda para la barra de menú.
Insets getMargin()	Obtiene el margen entre el borde de la barra de menú y sus menús.
JMenu getMenu(int index)	Obtiene el menú en la posición indicada en la barra de menú.
int getMenuCount()	Obtiene el número de elementos en la barra de menú.
SingleSelectionModel getSelectionModel()	Obtiene el modelo de objetos que maneja las selecciones simples.
MenuElement[] getSubElements()	Devuelve los menús en esta barra de menú.

Método	Descripción
MenuBarUI getUI()	Obtiene la IU actual de la barra de menú.
String getUIClassID()	Obtiene el nombre de la clase de apariencia que renderiza este componente.
boolean isBorderPainted()	Devuelve true si el borde de la barra de menú debe pintarse.
boolean isManagingFocus()	Devuelve true para indicar que este componente procesa los eventos de foco internamente.
boolean isSelected()	Devuelve true si la barra de menú tiene un componente seleccionado.
void menuSelectionChanged (boolean isIncluded)	Implementado para ser MenuElement pero actualmente no hace nada.
protected void paintBorder (Graphics g)	Pinta el borde de la barra de menú.
protected String paramString()	Obtiene una representación de cadena de esta JMenuBar.
void processKeyEvent (KeyEvent e, MenuElement[] path, MenuSelectionManager manager)	Implementado para ser MenuElement pero no hace nada.
void processMouseEvent (Mouse Event event, MenuElement[] path, MenuSelectionManager manager)	Implementado para ser MenuElement pero no hace nada.
void removeNotify()	Sobreescribe JComponent.removeNotify para eliminar la barra de menú del registro.
void setBorderPainted(boolean s)	Indica si el borde debería pintarse.
void setHelpMenu(JMenu menu)	Asigna el camino de ayuda que aparece cuando el usuario selecciona la opción Ayuda en la barra de menú.
void setMargin(Insets margin)	Asigna el margen entre el borde de la barra de menú y sus menús.
void setSelected(Component sel)	Asigna el componente seleccionado.
void setSelectionModel (SingleSelectionModel model)	Asigna el modelo de objetos para procesar selecciones simples.

Haremos funcionar JMenuBar a lo largo de los próximos temas mientras creamos un sistema de menús básico. El siguiente paso es crear los menús para añadir a la barra de menús y lo haremos en el tema siguiente.

Crear un menú

El programador novato vuelve y dice, "He creado una barra de menús, pero no aparece nada en ella; ¿qué sucede?". Sonreímos y decimos, "Tiene que añadir los menús explícitamente". "Ah", dice el PN, "muéstreme ahora cómo funciona".

Creamos los menús de una barra de menús con la clase JMenu. Aquí tiene el diagrama de herencia para esa clase (observe que JMenu es realmente una subclase de JMenuItem, debido a que JMenuItem, que a su vez subclasifica a AbstractButton, puede responder a clics de ratón):

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.AbstractButton
                          |
                          +-javax.swing.JMenuItem
                                |
                                +-javax.swing.JMenu
```

La tabla 16.3 muestra el campo de la clase JMenu, la tabla 16.4 muestra sus constructores y la tabla 16.5 sus métodos.

Tabla 16.3. El campo de la clase JMenu.

Campo	Descripción
protected JMenu.WinListener popupListener	El receptor de cierre de ventana emergente.

Tabla 16.4. Constructores de la clase JMenu.

Constructor	Descripción
JMenu()	Construye un nuevo JMenu.
JMenu(String s)	Construye un nuevo JMenu con la cadena como texto.

Constructor	Descripción
JMenu(String s, boolean b)	Construye un nuevo JMenu con la cadena como texto y especifica si tiene un menú *roto*, como indica el valor *boolean*.

Tabla 16.5. Métodos de la clase JMenu.

Método	Descripción
JMenuItem add(Action a)	Construye un nuevo elemento de menú añadido al objeto Action indicado y la añade al final de este menú.
Component add(Component c)	Añade un componente al final de este menú.
JMenuItem add(JMenuItem menuItem)	Añade un elemento de menú al final de este menú.
JMenuItem add(String s)	Construye un nuevo elemento de menú con el texto indicado y lo añade al final de este menú.
void addMenuListener (MenuListener l)	Añade un receptor para eventos de menú.
void addSeparator()	Añade un nuevo separador al final del menú.
protected PropertyChange Listener createActionChange-Listener(JMenuItem b)	Crea un receptor de cambios de acción.
protected JMenu.WinListener createWinListener(JPopupMenu p)	Crea un receptor de cierre de ventana.
void doClick(int pressTime)	Realiza una acción de clic programáticamente.
protected void fireMenuCanceled()	Notifica a todos los receptores registrados para esta notificación que ha ocurrido este evento.
protected void fireMenu-Deselected()	Notifica a todos los receptores registrados para esta notificación.
protected void fireMenuSelected()	Notifica a todos los receptores registrados para esta notificación.

Método	Descripción
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
Component getComponent()	Devuelve el java.awt.Component utilizado para pintar este MenuElement.
int getDelay()	Obtiene el retraso sufrido antes de que se muestren o cierren los menús emergentes.
JMenuItem getItem(int pos)	Obtiene el JMenuItem en la posición indicada.
int getItemCount()	Obtiene el número de elementos de menú, incluyendo separadores.
Component getMenu-Component(int n)	Obtiene el componente que ocupa la posición n.
int getMenuComponentCount()	Obtiene el número de componentes del menú.
Component[] getMenuComponents()	Obtiene una matriz con los componentes de menú.
JPopupMenu getPopupMenu()	Obtiene el menú emergente asociado con este menú.
MenuElement[] getSubElements()	Obtiene la matriz que contiene los componentes submenús para este componente menú.
String getUIClassID()	Obtiene la clase de apariencia que renderiza este componente.
JMenuItem insert(Action a, int pos)	Inserta un nuevo elemento de menú añadido al objeto Action indicado en la posición nada.
JMenuItem insert (JMenuItem mi, int pos)	Inserta el JMenuitem indicado en una posición dada.
void insert(String s, int pos)	Inserta un nuevo elemento de menú con el texto indicado en una posición dada.
void insertSeparator(int index)	Inserta un separador en una posición indicada.
boolean isMenuComponent (Component c)	Devuelve true si el componente indicado existe en la jerarquía de submenús.

Método	Descripción
boolean isPopupMenuVisible()	Devuelve true si en la ventana de menú emergente es visible.
boolean isSelected()	Devuelve true si el menú está seleccionado en este momento (emergente).
boolean isTearOff()	Devuelve true si el menú se puede desactivar.
boolean isTopLevelMenu()	Devuelve true si el menú es un menú de primer nivel.
void menuSelectionChanged (boolean isIncluded)	Se llama cuando la selección de la barra de menú cambia.
protected String paramString()	Obtiene una representación de cadena de este JMenu.
protected void processKey Event(KeyEvent e)	Sobrescribe processKeyEvent para procesar eventos.
void remove(Component c)	Elimina el componente.
void remove(int pos)	Elimina el elemento de menú en la posición dada.
void remove(JMenuItem item)	Elimina el elemento de menú indicado.
void removeAll()	Elimina todos los elementos de menú a partir de este menú.
void removeMenuListener (MenuListener l)	Elimina un receptor para eventos de menú.
void setAccelerator (KeyStroke keyStroke)	No definido para JMenu.
void setDelay(int d)	Asigna el retraso sugerido antes de que se muestre o cierre un menú emergente.
void setMenuLocation(int x, int y)	Asigna la localización del componente emergente.
void setModel(ButtonModel newModel)	Asigna el modelo de datos para el botón de menú.
void setPopupMenuVisible (boolean b)	Asigna la visibilidad de la porción emergente del menú.

Método	Descripción
void setSelected(boolean b)	Asigna el estado de selección del menú.
void updateUI()	Llamado por UIFactory cuando cambia la apariencia.

Cuando creamos los menús utilizando la clase JMenu, poblamos estos menús con elementos de menús (consulte el siguiente tema para ver los detalles).

Crear un elemento de menú

"Perfecto", dice el programador novato, "he creado algunos menús. ¿Cómo añado elementos de menú a estos menús?; ¿con la clase JMenuItem? ". "Exacto", decimos. Los elementos reales en los menús de la Swing están soportados por la clase JMenuItem. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.AbstractButton
                          |
                          +-javax.swing.JMenuItem
```

La tabla 16.6 muestra los constructores de la clase JMenuItem y la tabla 16.7 sus métodos.

Tabla 16.6. Constructores de la clase JMenuItem.

Constructor	Descripción
JMenuItem()	Construye un JMenuItem.
JMenuItem(Icon icon)	Construye un JMenuItem con un icono.
JMenuItem(String text)	Construye un JMenuItem con texto.
JMenuItem(String text, Icon icon)	Construye un JMenuItem con el texto proporcionado y un icono.
JMenuItem(String text, int mnemonic)	Construye un JMenuItem con el texto y mnemónico de teclado proporcionados.

Tabla 16.7. Métodos de la clase JMenuItem.

Método	Descripción
void addMenuDragMouse Listener(MenuDragMouse Listener l)	Añade un MenuDragMouseListener.
void addMenuKeyListener (MenuKeyListener l)	Añade un MenuKeyListener al elemento de menú.
protected void fireMenuDrag MouseDragged(MenuDragMouse Event event)	Dispara el evento "ratón arrastrado".
protected void fireMenuDrag MouseEntered(MenuDragMouse Event event)	Dispara el evento "ratón entra".
protected void fireMenuDrag MouseExited(MenuDragMouse Event event)	Dispara el evento "ratón sale".
protected void fireMenuDrag MouseReleased(MenuDragMouse Event event)	Dispara el evento "ratón suelto".
protected void fireMenuKey Pressed(MenuKeyEvent event)	Dispara el evento "tecla pulsada".
protected void fireMenuKey Released(MenuKeyEvent event)	Dispara el evento "tecla soltada".
protected void fireMenuKey Typed(MenuKeyEvent event)	Dispara el evento "Introduce tecla".
KeyStroke getAccelerator()	Obtiene el KeyStroke que sirve como acelerador para el elemento de menú.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext asociado con este JComponent.
Component getComponent()	Devuelve el java.awt.Component utilizado para pintar este objeto.
MenuElement[] getSubElements()	Devuelve una matriz que contiene los componentes de submenú para este componente menú.
String getUIClassID()	Obtiene el nombre de la clase de apariencia que renderiza este componente.

Método	Descripción
protected void init(String text, Icon icon)	Inicia el elemento de menú con el texto e icono indicados.
boolean isArmed()	Determina si el elemento de menú está armado.
void menuSelectionChanged (boolean isIncluded)	Llamada por MenuSelectionManager cuando cambia el estado de selección de MenuElement.
protected String paramString()	Obtiene una representación de cadena de este JMenuItem.
void processKeyEvent (KeyEvent e, MenuElement[] path, MenuSelectionManager manager)	Procesa un evento de tecla procedente del MenuSelectionManager.
void processMenuDragMouse Event(MenuDragMouseEvent e)	Maneja el arrastre del ratón en un menú.
void processMenuKeyEvent (MenuKeyEvent e)	Maneja una pulsación de tecla en un menú.
void processMouseEvent (MouseEvent e, MenuElement[] path, MenuSelectionManager manager)	Procesa un evento de ratón procedente del MenuSelectionManager.
void removeMenuDragMouse Listener(MenuDragMouse Listener l)	Elimina un MenuDragMouseListener.
void removeMenuKeyListener (MenuKeyListener l)	Elimina un MenuKeyListener del elemento de menú.
void setAccelerator (KeyStroke keyStroke)	Asigna la combinación de teclas que invoca a los receptores de la acción del elemento de menú sin navegar por la jerarquía de menús.
void setArmed(boolean b)	Identifica como armado al elemento menú.
void setEnabled(boolean b)	Habilita o inhabilita el elemento de menú.
void setUI(MenuItemUI ui)	Asigna el objeto apariencia que renderiza este componente.
void updateUI()	Llamada por UIFactory cuando cambia la apariencia.

Como muestran las tablas 16.6 y 16.7, la clase JMenuItem tiene mucho que ofrecer; por ejemplo, puede determinar si un elemento de menú esta "armado" (es decir, quedará seleccionado si el usuario suelta el botón del ratón) con los métodos isArmed y setArmed.

Este tema y los dos anteriores han introducido las barras de menús, menús y elementos de menú en la Swing; conjuntaremos estos elementos en el siguiente tema para crear un sistema de menús básico.

Crear un sistema de menús básico

"He creado un objeto de barra de menús, objetos menús y objetos elemento de menú", dice el programador novato, "pero, ¿cómo diablos los uno?". "Siéntese y lo haremos", decimos. "No es difícil".

Los tres temas anteriores han introducido las clases necesarias para crear menús en la programación Swing: JMenuBar, JMenu y JMenuItem. Haremos funcionar estas clases en un ejemplo que visualiza dos menús, **Archivo** y **Edición**, en una barra de menús. Cuando el usuario hace clic sobre un elemento de menú, el código visualizará el elemento seleccionado en la barra de estado del applet.

Vamos a comenzar creando una barra de menús con la clase JMenuBar. Después, crearemos el menú **Archivo** con la clase JMenu y los tres elementos de menú para ese menú: **Nuevo**, **Abrir** y **Salir**, utilizando la clase JMenuItem:

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

    /*
  <APPLET
    CODE = menu.class
    WIDTH = 350
    HEIGHT = 280 >
  </APPLET>
    */

public class menu extends JApplet implements ActionListener
{
    public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu1 = new JMenu("Archivo");
        JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
            jmenuitem2 = new JMenuItem("Abrir..."),
            jmenuitem3 = new JMenuItem("Salir");
```

```
    ...
}
```

Ahora, podemos añadir los tres elementos de menú al menú **Archivo** utilizando el método add de la clase JMenu. Con los elementos de menú se utilizan receptores de acción, por lo que proporcionaremos a cada elemento de menú un comando de acción y añadiremos un receptor de acción a cada elemento (observe que los separadores de menú se pueden añadir con el método addSeparator):

```
public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu1 = new JMenu("Archivo");
        JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
            jmenuitem2 = new JMenuItem("Abrir..."),
            jmenuitem3 = new JMenuItem("Salir");

        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
        jmenu1.add(jmenuitem3);

        jmenuitem1.setActionCommand("Seleccionó Nuevo");
        jmenuitem2.setActionCommand("Seleccionó Abrir");

        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);
        ...
```

Puede crear el menú **Edición** de la misma forma, proporcionándole tres elementos: **Copiar**, **Pegar** y **Cortar**:

```
public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu1 = new JMenu("Archivo");
        JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
            jmenuitem2 = new JMenuItem("Abrir..."),
            jmenuitem3 = new JMenuItem("Salir");

        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
        jmenu1.add(jmenuitem3);

        jmenuitem1.setActionCommand("Seleccionó Nuevo");
        jmenuitem2.setActionCommand("Seleccionó Abrir");

        jmenuitem1.addActionListener(this);
```

```
jmenuitem2.addActionListener(this);

JMenu jmenu2 = new JMenu("Edición");
JMenuItem jmenuitem4 = new JMenuItem("Cortar"),
    jmenuitem5 = new JMenuItem("Copiar"),
    jmenuitem6 = new JMenuItem("Pegar");

jmenu2.add(jmenuitem4);
jmenu2.add(jmenuitem5);
jmenu2.add(jmenuitem6);

jmenuitem4.setActionCommand("Seleccionó Cortar");
jmenuitem5.setActionCommand("Seleccionó Copiar");
jmenuitem6.setActionCommand("Seleccionó Pegar");

jmenuitem4.addActionListener(this);
jmenuitem5.addActionListener(this);
jmenuitem6.addActionListener(this);
...
```

Lo único que queda por hacer es utilizar el método add de la clase JMenuBar para añadir elementos de menú a la barra de menús y para añadir la barra de menú al applet con el método setJMenuBar:

```
public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu1 = new JMenu("Archivo");
        JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
            jmenuitem2 = new JMenuItem("Abrir..."),
            jmenuitem3 = new JMenuItem("Salir");

        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
        jmenu1.add(jmenuitem3);

        jmenuitem1.setActionCommand("Seleccionó Nuevo");
        jmenuitem2.setActionCommand("Seleccionó Abrir");

        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);

        JMenu jmenu2 = new JMenu("Edición");
        JMenuItem jmenuitem4 = new JMenuItem("Cortar"),
            jmenuitem5 = new JMenuItem("Copiar"),
            jmenuitem6 = new JMenuItem("Pegar");

        jmenu2.add(jmenuitem4);
        jmenu2.add(jmenuitem5);
        jmenu2.add(jmenuitem6);

        jmenuitem4.setActionCommand("Seleccionó Cortar");
```

```
        jmenuitem5.setActionCommand("Seleccionó Copiar");
        jmenuitem6.setActionCommand("Seleccionó Pegar");

        jmenuitem4.addActionListener(this);
        jmenuitem5.addActionListener(this);
        jmenuitem6.addActionListener(this);

        jmenubar.add(jmenu1);
        jmenubar.add(jmenu2);

        setJMenuBar(jmenubar);
    }
```

Ahora mostramos el sistema de menús al usuario. Cuando el usuario hace clic sobre un elemento de menú, obtendremos el comando action del elemento y lo visualizaremos en la barra de estado del applet utilizando el método actionPerformed. Aquí tiene el código:

```
public void actionPerformed(ActionEvent e)
    {
        JMenuItem jmenuitem = (JMenuItem)e.getSource();

        showStatus(jmenuitem.getActionCommand());
    }
```

Esto fue todo lo necesario para crear un sistema de menús básico. Trabajar con menús en la Swing es parecido a trabajar con ellos en AWT, con la diferencia obvia de que los applet en AWT no pueden visualizar menús. El resultado de este código se muestra en la figura 16.1, donde vemos el sistema de menús en funcionamiento. Este ejemplo (archivo menu.java en el CD) funciona exactamente como fue diseñado.

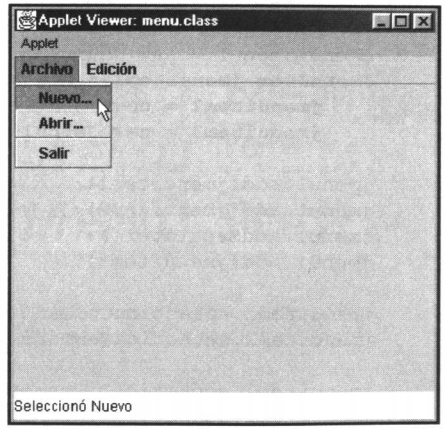


Figura 16.1. Un sistema de menús básico.

Adición de imágenes a menús

El coordinador de diseño estético dice, "Sus menús son pobres; ¿podría mejorar las cosas un poco?". "Bien", decimos, "puedo añadir imágenes". "¡Fantástico!", dice el CDE. "Le enviaré las imágenes que quiero utilizar; son sólo de 1.024 por 1.024 *pixels*". "Ah", decimos.

Es sencillo añadir imágenes a menús en la programación Swing; basta con añadir un icono a la llamada al constructor adecuado. Aquí tiene cómo se realiza la adición de imágenes a cada elemento de menú en el ejemplo del tema anterior:

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

    /*
 <APPLET
    CODE = menuimages.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class menuimages extends JApplet implements ActionListener
{

    ImageIcon icon = new ImageIcon("item.jpg");

    public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu1 = new JMenu("Archivo");
        JMenuItem jmenuitem1 = new JMenuItem("Nuevo...", icon),
            jmenuitem2 = new JMenuItem("Abrir...", icon),
            jmenuitem3 = new JMenuItem("Salir", icon);

        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
        jmenu1.add(jmenuitem3);

        jmenuitem1.setActionCommand("Seleccionó Nuevo");
        jmenuitem2.setActionCommand("Seleccionó Abrir");

        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);

        JMenu jmenu2 = new JMenu("Edición");
```

```
        JMenuItem jmenuitem4 = new JMenuItem("Cortar", icon),
            jmenuitem5 = new JMenuItem("Copiar", icon),
            jmenuitem6 = new JMenuItem("Pegar", icon);

        jmenu2.add(jmenuitem4);
        jmenu2.add(jmenuitem5);
        jmenu2.add(jmenuitem6);

        jmenuitem4.setActionCommand("Seleccionó Cortar");
        jmenuitem5.setActionCommand("Seleccionó Copiar");
        jmenuitem6.setActionCommand("Seleccionó Pegar");

        jmenuitem4.addActionListener(this);
        jmenuitem5.addActionListener(this);
        jmenuitem6.addActionListener(this);

        jmenubar.add(jmenu1);
        jmenubar.add(jmenu2);

        setJMenuBar(jmenubar);
    }

    public void actionPerformed(ActionEvent e)
    {
        JMenuItem jmenuitem = (JMenuItem)e.getSource();

        showStatus(jmenuitem.getActionCommand());
    }
}
```

La figura 16.2 muestra el resultado. Como vemos, cada elemento de menú presenta ahora una imagen.

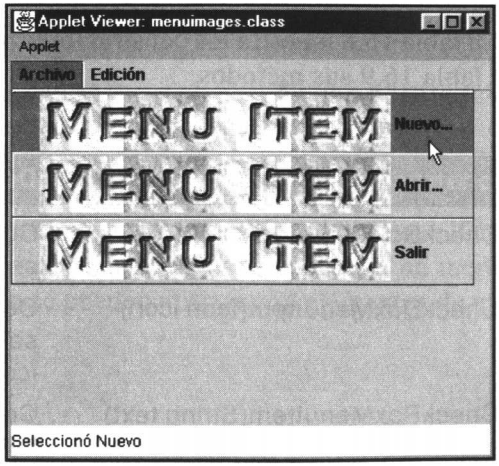


Figura 16.2. Adición de imágenes a menús.

Crear elementos de menú de casillas de verificación

El especialista de soporte a productos aparece y dice, "Los usuarios no están contentos con nuestro nuevo programa, especialmente el elemento de menú Hacer todo el texto invisible, debido a que nunca saben si esa opción está habilitada hasta que ya es demasiado tarde". "Bien", decimos, "incluiremos una casilla verificación sobre el elemento para mostrar cuándo está activo".

Los menús de Swing soportan elementos de menú de casillas de verificación como lo hacían los menús AWT. Puede utilizar la clase JCheckBoxMenuItem para crear elementos de menú de casilla de verificación en la programación Swing. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.AbstractButton
                          |
                          +-javax.swing.JMenuItem
                                |
                                +-javax.swing.JCheckBoxMenuItem
```

La tabla 16.8 muestra los constructores para la clase JCheckBoxMenuItem y la tabla 16.9 sus métodos.

Tabla 16.8. Constructores de la clase JCheckBoxMenuItem.

Constructor	Descripción
JCheckBoxMenuItem()	Construye un checkboxMenuItem no seleccionado.
JCheckBoxMenuItem(Icon icon)	Construye un checkboxMenuItem no seleccionado inicialmente con un icono.
JCheckBoxMenuItem(String text)	Construye un checkboxMenuItem no seleccionado inicialmente con texto.
JCheckBoxMenuItem(String text, boolean b)	Construye un checkboxMenuItem con el texto y estado indicados.

Constructor	Descripción
JCheckBoxMenuItem(String text, Icon icon)	Construye un checkboxMenuItem no seleccionado inicialmente con el texto e icono indicados.
JCheckBoxMenuItem(String text, Icon icon, boolean b)	Construye un checkboxMenuItem con el estado, texto e icono indicados.

Tabla 16.9. Métodos de la clase JCheckBoxMenuItem.

Método	Descripción
AccessibleContext getAccessible-Context()	Obtiene el AccessibleContext.
Object[] getSelectedObjects()	Obtiene una matriz (longitud 1) que contiene la etiqueta del elemento de menú de la casilla de verificación.
boolean getState()	Obtiene el estado seleccionado del elemento.
String getUIClassID()	Obtiene el nombre de la clase de apariencia que renderiza éste componente.
protected String paramString()	Obtiene una cadena de este JCheck-BoxMenuItem.
void requestFocus()	Sobreescribe JComponent.request-Focus() para impedir la pérdida del foco.
void setState(boolean b)	Asigna el estado seleccionado del elemento.

Vamos a examinar un ejemplo. En este caso, añadimos cuatro elementos de casillas de verificación a un menú utilizando la clase JCheckBoxMenuItem. Aquí tiene cómo crear y añadir estos elementos a un menú (observe que los receptores de acción se pueden utilizar con elementos de menús de casillas de verificación):

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

    /*
 <APPLET
```

```
    CODE = menucheckbox.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class menucheckbox extends JApplet implements ActionListener
{
    ImageIcon icon = new ImageIcon("item.jpg");

    JCheckBoxMenuItem
       jcheckboxmenuitem1 = new JCheckBoxMenuItem("Elemento 1", icon),
       jcheckboxmenuitem2 = new JCheckBoxMenuItem("Elemento 2", icon),
       jcheckboxmenuitem3 = new JCheckBoxMenuItem("Elemento 3", icon),
       jcheckboxmenuitem4 = new JCheckBoxMenuItem("Elemento 4", icon);

    public void init()
    {
        Container contentPane = getContentPane();

        JMenuBar jmenubar = new JMenuBar();
          JMenu jmenu = new JMenu("Elementos de menú de casilla de
verificación");

        jcheckboxmenuitem1.addActionListener(this);
        jcheckboxmenuitem2.addActionListener(this);
        jcheckboxmenuitem3.addActionListener(this);
        jcheckboxmenuitem4.addActionListener(this);

        jmenu.add(jcheckboxmenuitem1);
        jmenu.add(jcheckboxmenuitem2);
        jmenu.add(jcheckboxmenuitem3);
        jmenu.add(jcheckboxmenuitem4);

        jmenubar.add(jmenu);
        setJMenuBar(jmenubar);
    }
    ...
}
```

Cuando el usuario hace clic sobre un elemento de menú de casilla de verificación, mostramos el estado de los cuatro elementos, como sigue:

```
public void actionPerformed(ActionEvent e)
    {
        showStatus("Elemento 1:  " + jcheckboxmenuitem1.getState() +
            " Elemento 2:  " + jcheckboxmenuitem2.getState() +
            " Elemento 3:  " + jcheckboxmenuitem3.getState() +
            " Elemento 4:  " + jcheckboxmenuitem4.getState());
    }
```

Es todo lo que hace falta. La figura 16.3 muestra el resultado de este código. Como muestra la figura, las casillas de verificación aparecen como

deberían en el menú. Este ejemplo se encuentra en el archivo menucheckbox.java del CD.

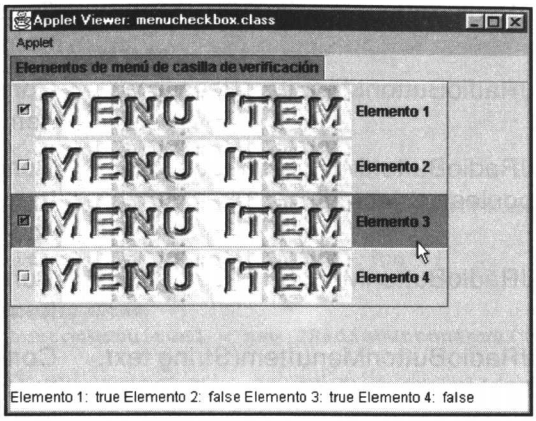


Figura 16.3. Creación de elementos de menú de casilla de verificación.

Crear menús de botones de activación

"Bien", dice el programador novato, "conozco la forma de crear elementos de menús de casilla de verificación con la clase JCheckBoxMenuItem. Pero, ¿qué sucede con los botones de activación? ¿No tenemos que añadir elementos de casillas de verificación a un grupo?". "Casi es correcto", decimos. "De hecho, añadimos objetos JRadioButtonMenuItem a un grupo". Utilizamos la clase JRadioButtonMenuItem para crear elementos de menú de botones de activación en la Swing. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.AbstractButton
                          |
                          +-javax.swing.JMenuItem
                                |
                                +-javax.swing.JRadioButtonMenuItem
```

La tabla 16.10 muestra los constructores de la clase JRadioButtonMenuItem y la tabla 16.11 sus métodos.

Tabla 16.10. Constructores de la clase JRadioButtonMenuItem.

Constructor	Descripción
JRadioButtonMenuItem()	Construye un JRadioButtonMenuItem.
JRadioButtonMenuItem(Icon icon)	Construye un JRadioButtonMenuItem con un icono.
JRadioButtonMenuItem(Icon icon, boolean selected)	Construye un JRadioButtonMenuItem con el estado de selección e imagen indicados, pero sin texto.
JRadioButtonMenuItem(String text)	Construye un JRadioButtonMenuItem con texto.
JRadioButtonMenuItem(String text, boolean b)	Construye un elemento de menú de botón de activación con el texto indicado y el estado de selección.
JRadioButtonMenuItem(String text, Icon icon)	Construye un JRadioButtonMenuItem con el texto e icono indicados.
JRadioButtonMenuItem(String text, Icon icon, boolean selected)	Construye un elemento de menú de botón de activación que tiene texto, imagen de icono y estado de selección indicados.

Tabla 16.11. Métodos de la clase JRadioButtonMenuItem.

Método	Descripción
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
String getUIClassID()	Obtiene el nombre de la clase de apariencia que renderiza este componente.
protected String paramString()	Obtiene una representación de cadena de este JRadioButtonMenuItem.
void requestFocus()	Sobrescribe Component.requestFocus() para no perder el foco.

Vamos a examinar un ejemplo. En este caso, añadimos cuatro botones de activación a un menú y también un receptor de elementos para indicar cuándo está seleccionado o deseleccionado cada botón de activación. Aquí tiene la apariencia del código:

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

    /*
 <APPLET
    CODE = menuradiobutton.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class menuradiobutton extends JApplet implements ItemListener
{
    ImageIcon icon = new ImageIcon("item.jpg");

    JRadioButtonMenuItem
        jradiobuttonmenuitem1 = new JRadioButtonMenuItem("Elemento 1",
icon),
        jradiobuttonmenuitem2 = new JRadioButtonMenuItem("Elemento 2",
icon),
        jradiobuttonmenuitem3 = new JRadioButtonMenuItem("Elemento 3",
icon),
        jradiobuttonmenuitem4 = new JRadioButtonMenuItem("Elemento 4",
icon);

    public void init()
    {
        Container contentPane = getContentPane();

        JMenuBar jmenubar = new JMenuBar();
            JMenu jmenu = new JMenu("Elementos de menú de botón de
activación");

        jmenu.add(jradiobuttonmenuitem1);
        jmenu.add(jradiobuttonmenuitem2);
        jmenu.add(jradiobuttonmenuitem3);
        jmenu.add(jradiobuttonmenuitem4);

        ButtonGroup group = new ButtonGroup();
        group.add(jradiobuttonmenuitem1);
        group.add(jradiobuttonmenuitem2);
        group.add(jradiobuttonmenuitem3);
        group.add(jradiobuttonmenuitem4);

        jradiobuttonmenuitem1.addItemListener(this);
        jradiobuttonmenuitem2.addItemListener(this);
        jradiobuttonmenuitem3.addItemListener(this);
        jradiobuttonmenuitem4.addItemListener(this);

        jmenubar.add(jmenu);
        setJMenuBar(jmenubar);
    }

    public void itemStateChanged(ItemEvent e)
```

```
    {
        JMenuItem jmenuitem = (JMenuItem) e.getSource();
        String itemtext = jmenuitem.getText();

        if(e.getStateChange() == ItemEvent.SELECTED)
            itemtext += " seleccionado";
        else
            itemtext += " deseleccionado";

        showStatus(itemtext);
    }
}
```

La figura 16.4 muestra el resultado de este código, donde se muestra el funcionamiento en el menú de los botones de activación. Cuando el usuario hace clic sobre un botón de activación, este applet indica cuál se seleccionó en la barra de estado. Este ejemplo (archivo menuradiobutton.java en el CD) es un éxito.

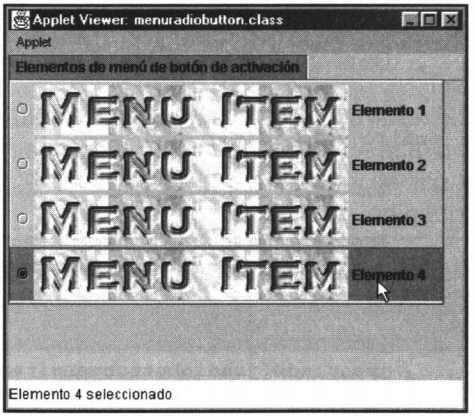


Figura 16.4. Creación de elementos de menú de botones de activación.

Crear submenús

"Mis menús son muy grandes", dice el programador novato. "Quiero permitir al usuario seleccionar el color de dibujo en mi programa mediante elementos de menú, pero todos los colores están hacinando a los otros elementos del menú". "Debería probar a poner todos los colores en un submenú", decimos. "¿Cuántos colores quiere mostrar?". "Sobre 3.000", dice el PN. Decimos, "Vaya".

Los submenús, también llamados *menús de cascada*, son menús que abrimos desde otros menús. La Swing indica si un elemento de menú es realmen-

te un submenú añadiendo una flecha seleccionable a la derecha del elemento de menú; cuando el usuario hace clic sobre la fecha, se abre el submenú.

Puede crear menús de submenús fácilmente en la Swing. Lo único que hace falta es crear un nuevo menú, añadir elementos al menú y después añadir el mismo, como elemento en otro menú.

Aquí tiene un ejemplo que lo aclara. En este caso, puede añadir un submenú a un menú y proporcionar al submenú cuatro elementos. Aquí tiene la apariencia del código:

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

    /*
 <APPLET
    CODE = submenus.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class submenus extends JApplet
{

    public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu  = new JMenu("Sub Menús", true);
        JMenu jsubmenu    = new JMenu("Menú en cascada", true);

        jmenu.add("Elemento 1");
        jmenu.add("Elemento 2");
        jmenu.add("Elemento 3");
        jmenu.add("Elemento 4");

        jsubmenu.add("Sub Elemento 1");
        jsubmenu.add("Sub Elemento 2");
        jsubmenu.add("Sub Elemento 3");
        jsubmenu.add("Sub Elemento 4");

        jmenu.add(jsubmenu);

        jmenubar.add(jmenu);

        setJMenuBar(jmenubar);
   }
}
```

La figura 16.5 muestra el resultado de este código. Como puede ver, el submenú se abre cuando el usuario selecciona el elemento de menú correspondiente. Este ejemplo se encuentra en el archivo submenus.java del CD.

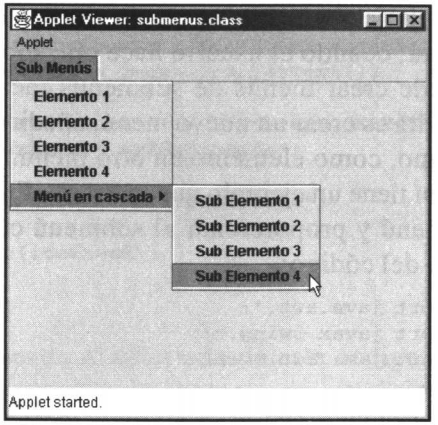


Figura 16.5. Creación de submenús.

Crear aceleradores y mnemónicos de menú

"Dígame", dice el zar de la corrección de la programación, "¿qué hay acerca de aquellos elementos de menú que a veces veo que tienen aceleradores de teclado? Apuesto a que no puede hacerlo en Java". "Seguro que se puede", decimos, "con mnemónicos de menús y aceleradores".

Los *aceleradores y mnemónicos de menú* proporcionan acceso de teclado a elementos de menú. Los mnemónicos están representados por caracteres subrayados en el texto de un menú o elemento de menú. Cuando el usuario pulsa la meta tecla para el sistema (como la tecla **Alt** en *Windows*) y la tecla de mnemónicos, se abre el elemento de menú. Puede asignar un mnemónico de menú o de elemento de menú con el método setMnemonic.

Los aceleradores son muy similares a los mnemónicos, excepto en que se especifican las pulsaciones de tecla reales necesarias para activar un elemento de menú, como **F1**, **Control+X** y así sucesivamente (la meta tecla no se utiliza en aceleradores de menú a menos que especifique que debiera ser así). Puede añadir un acelerador a un elemento de menú (no a menús) con el método setAccelerator, el cual recibe un objeto de la clase KeyStroke que define la pulsación de tecla que quiere utilizar como acelerador. Aquí tiene el diagrama de herencia para la clase KeyStroke:

```
java.lang.Object
  |
  +--javax.swing.KeyStroke
```

La tabla 16.12 muestra los métodos de la clase KeyStroke.

Puede utilizar las constantes de teclas definidas en la clase KeyEvent (KeyEvent.VK_A, KeyEvent.VK_ENTER y KeyEvent.VK_TAB) para especificar el código de tecla cuando crea un objeto KeyStroke. Los modificadores que puede especificar pueden ser cualquier combinación de los siguientes:

- Event.SHIFT_MASK (= 1)
- Event.CTRL_MASK (= 2)
- Event.META_MASK (= 4)
- Event.ALT_MASK (= 8)

Tabla 16.12. Métodos de la clase KeyStroke.

Método	Descripción
boolean equals(Object anObject)	Devuelve true si el objeto es idéntico al objeto indicado.
char getKeyChar()	Obtiene el carácter definido por este objeto KeyStroke.
int getKeyCode()	Obtiene el código numérico de tecla definido para este objeto KeyStroke.
static KeyStroke getKeyStroke(char keyChar)	Devuelve una instancia compartida de pulsación de tecla que se activa cuando se pulsa la tecla.
static KeyStroke getKeyStroke(char keyChar, boolean onKeyRelease)	Obsoleto. Utilice getKeyStroke(char).
static KeyStroke getKeyStroke(int keyCode, int modifiers)	Devuelve una instancia compartida de una pulsación de tecla dado un código de carácter y un conjunto de modificadores. La tecla está activada cuando se pulsa.
static KeyStroke getKeyStroke(int keyCode, int modifiers, boolean onKeyRelease)	Obtiene una instancia compartida de una pulsación de tecla dado un código de tecla numérico y un conjunto de modificadores.
static KeyStroke getKeyStroke(String representation)	Analiza una cadena y devuelve un KeyStroke.
static KeyStroke getKeyStrokeForEvent(KeyEvent anEvent)	Devuelve una pulsación de tecla a partir de un evento.

Método	Descripción
int getModifiers()	Obtiene las teclas modificadoras definidas para este objeto KeyStroke.
int hashCode()	Obtiene un valor numérico para este objeto como valor de índice en una tabla *hash*.
boolean isOnKeyRelease()	Devuelve true si esta pulsación de tecla está activa en el momento de soltar la tecla.
String toString()	Obtiene una cadena que visualiza e identifica las propiedades del objeto.

Lo veremos todo en un ejemplo. En este caso, puede añadir un mnemónico, la letra *N* al elemento **Nuevo** en el menú **Archivo** y asignar el acelerador **Control+N**. Aquí tiene la apariencia del código:

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

    /*
 <APPLET
    CODE = menuaccelerator.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class menuaccelerator extends JApplet implements ActionListener
{
    public void init()
    {
        Container contentPane = getContentPane();

        JMenuBar jmenubar = new JMenuBar();
        JMenu jmenu = new JMenu("Archivo");
        JMenuItem jmenuitem = new JMenuItem("Nuevo...");

        jmenu.add(jmenuitem);
        jmenu.add("Abrir ...");
        jmenu.addSeparator();
        jmenu.add("Salir");

        jmenuitem.setMnemonic(KeyEvent.VK_N);

          KeyStroke keystroke = KeyStroke.getKeyStroke(KeyEvent.VK_N,
Event.CTRL_MASK);
```

```
            jmenuitem.setAccelerator(keystroke);

            jmenuitem.addActionListener(this);

            jmenubar.add(jmenu);
            setJMenuBar(jmenubar);
        }

        public void actionPerformed(ActionEvent e)
        {
            showStatus("Seleccionó el elemento Nuevo.");
        }
    }
```

La figura 16.6 muestra el resultado, donde puede ver tanto el mnemónico (la *N* subrayada en el elemento de menú **Nuevo**) como el acelerador (**Control+N**). Proporcionar mnemónicos y aceleradores como éstos puede acelerar las cosas para los usuarios que no quieran alternar entre el teclado y el ratón una y otra vez. Este ejemplo se encuentra en el archivo menuaccelerator.java del CD.

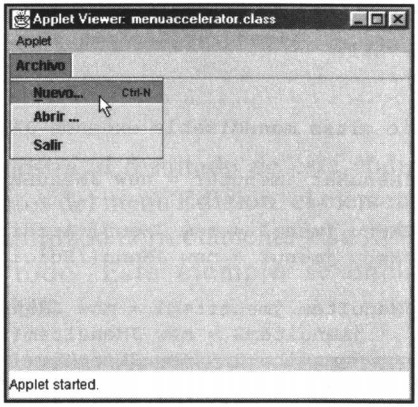


Figura 16.6. Mnemónicos y aceleradores de menú.

Habilitar/inhabilitar elementos de menú y cambiar títulos en tiempo de ejecución

"Ese maldito Felipe", dice el programador novato, "hizo clic con el ratón sobre la opción incorrecta en mi programa en el momento inadecuado, lo que hizo que mi código intentara comprobar el correo electrónico cuando ni siquiera estaba conectado a la red Internet". "Vaya con Felipe", decimos. "¿Qué tal si inhabilitamos los elementos de menú cuando no sean adecuados?". "¡Perfecto!" dice el PN.

Puede habilitar o inhabilitar elementos de menú con el método setEnabled; cuando un elemento de menú está inhabilitado, aparece atenuado y no se puede pulsar. Aquí tiene un ejemplo rápido en el que permitimos que el usuario habilite un elemento de menú en el menú **Edición** de un applet. Para hacerlo, añada otros elementos al menú **Edición**: **Inhabilitar elemento inferior** y **Habilitar elemento inferior**. Cuando el usuario hace clic sobre **Inhabilitar elemento inferior**, inhabilitamos un tercer elemento de menú en el menú **Edición** y cambiamos su texto a "Elemento inhabilitado" con el método setText; cuando el usuario hace clic sobre el elemento **Habilitar elemento inferior**, habilitamos un tercer elemento y cambiamos su texto a "elemento habilitado". Aquí tiene el código:

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

    /*
 <APPLET
    CODE = menudisable.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class menudisable extends JApplet implements ActionListener
{
    JMenuBar jmenubar = new JMenuBar();

    JMenu jmenu1 = new JMenu("Archivo");
    JMenu jmenu2 = new JMenu("Edición");

    JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
        jmenuitem2 = new JMenuItem("Abrir..."),
        jmenuitem3 = new JMenuItem("Salir"),
        jmenuitem4 = new JMenuItem("Inhabilitar botón inferior"),
        jmenuitem5 = new JMenuItem("Habilitar botón inferior"),
        jmenuitem6 = new JMenuItem("Elemento habilitado");

    public void init()
    {
        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
        jmenu1.add(jmenuitem3);

        jmenuitem1.setActionCommand("Seleccionó Nuevo");
        jmenuitem2.setActionCommand("Seleccionó Abrir");

        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);

        jmenu2.add(jmenuitem4);
```

```
        jmenu2.add(jmenuitem5);
        jmenu2.add(jmenuitem6);

        jmenuitem4.setActionCommand("Seleccionó Cortar");
        jmenuitem5.setActionCommand("Seleccionó Copiar");

        jmenuitem4.addActionListener(this);
        jmenuitem5.addActionListener(this);

        jmenubar.add(jmenu1);
        jmenubar.add(jmenu2);

        setJMenuBar(jmenubar);
    }

    public void actionPerformed(ActionEvent e)
    {
        JMenuItem jmenuitem = (JMenuItem)e.getSource();
        if(jmenuitem == jmenuitem4) {
            jmenuitem6.setText("Elemento inhabilitado");
            jmenuitem6.setEnabled(false);
        }
        if(jmenuitem == jmenuitem5) {
            jmenuitem6.setText("Elemento habilitado");
            jmenuitem6.setEnabled(true);
        }
    }
}
```

La figura 16.7 muestra el resultado de este código. Cuando el usuario selecciona los elementos del menú **Edición**, el elemento inferior en ese menú está habilitado o inhabilitado correctamente y su título se cambia según sea necesario. Eso es todo. Este ejemplo se encuentra en el archivo menudisable.java del CD.

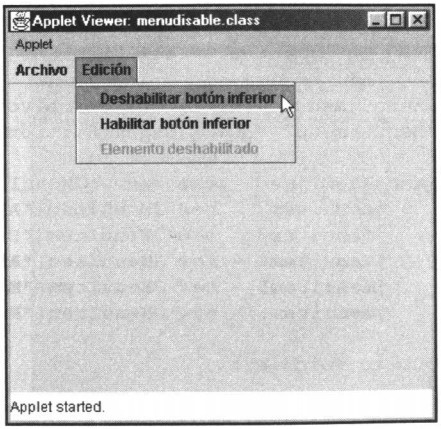


Figura 16.7. Inhabilitar elementos de menú.



Truco: No debería inhabilitar demasiados elementos de menús a la vez; hacerlo proporciona a su programa una apariencia inaccesible.

Añadir y eliminar elementos de menú en tiempo de ejecución

El gran jefe dice, "Estamos basando las opciones disponibles en nuestro nuevo programa en la cantidad que paga el usuario". Preguntamos: "¿Qué?". "Una vez hayamos determinado cuánto dinero tiene el usuario, habilitaremos el número adecuado de elementos de menú". "¿Qué?", preguntaremos.

Puede añadir elementos de menú a los menús en tiempo de ejecución con el método add y eliminarlos con el método remove. Aquí tiene un ejemplo en el que añadimos un nuevo elemento de menú a un menú cuando el usuario hace clic sobre **Añadir elemento** y lo eliminamos cuando el usuario hace clic sobre **Eliminar elemento**. Aquí tiene el código:

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

    /*
 <APPLET
    CODE = menuupdate.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class menuupdate extends JApplet implements ActionListener
{
    JMenuBar jmenubar = new JMenuBar();

    JMenu jmenu1 = new JMenu("Archivo");
    JMenu jmenu2 = new JMenu("Edición");

    JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
        jmenuitem2 = new JMenuItem("Abrir..."),
        jmenuitem3 = new JMenuItem("Salir"),
        jmenuitem4 = new JMenuItem("Añadir elemento"),
        jmenuitem5 = new JMenuItem("Eliminar elemento"),
        jmenuitem6 = new JMenuItem("Nuevo elemento");

    public void init()
    {
        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
```

```
        jmenu1.add(jmenuitem3);

        jmenuitem1.setActionCommand("Seleccionó Nuevo");
        jmenuitem2.setActionCommand("Seleccionó Abrir");

        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);

        jmenu2.add(jmenuitem4);
        jmenu2.add(jmenuitem5);

        jmenuitem4.setActionCommand("Seleccionó Cortar");
        jmenuitem5.setActionCommand("Seleccionó Copiar");

        jmenuitem4.addActionListener(this);
        jmenuitem5.addActionListener(this);

        jmenubar.add(jmenu1);
        jmenubar.add(jmenu2);

        setJMenuBar(jmenubar);
    }

    public void actionPerformed(ActionEvent e)
    {
        JMenuItem jmenuitem = (JMenuItem)e.getSource();
        if(jmenuitem == jmenuitem4) {
            jmenu2.add(jmenuitem6);
        }
        if(jmenuitem == jmenuitem5) {
            jmenu2.remove(jmenuitem6);
        }
    }
}
```

La figura 16.8 muestra el resultado de este código, donde puede ver el nuevo elemento añadido cuando el usuario hace clic sobre **Añadir elemento**.

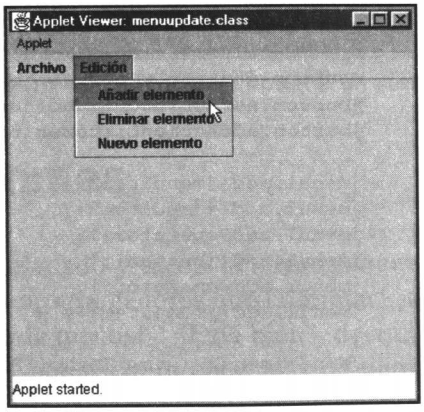


Figura 16.8. Añadir y eliminar elementos de menú en tiempo de ejecución.

El usuario puede eliminar el nuevo elemento haciendo clic sobre **Eliminar elemento**. Observe que puede crear nuevos elementos de menús en el momento con el operador new y puede añadir tantos elementos de menú a un menú como desee en tiempo de ejecución. Este ejemplo se encuentra en el archivo menuupdate.java del CD.

Añadir botones y otros controles a menús

Puede añadir controles como botones a menús de la Swing. De hecho, hacerlo es sencillo; basta con utilizar el método add de la clase JMenu. Por ejemplo, aquí tiene la forma de añadir un objeto JButton a un menú, haciendo que el código muestre un mensaje cuando el usuario hace clic sobre el botón:

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

    /*
  <APPLET
    CODE = menucontrols.class
    WIDTH = 350
    HEIGHT = 280 >
  </APPLET>
    */

public class menucontrols extends JApplet implements ActionListener
{
    public void init()
    {
        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu1 = new JMenu("Archivo");
        JMenuItem jmenuitem1 = new JMenuItem("Nuevo..."),
            jmenuitem2 = new JMenuItem("Abrir..."),
            jmenuitem3 = new JMenuItem("Salir");
        JButton jbutton = new JButton("¡Púlsame!");
        jbutton.setActionCommand("Ha hecho clic sobre el botón");
        jbutton.addActionListener(this);

        jmenu1.add(jmenuitem1);
        jmenu1.add(jmenuitem2);
        jmenu1.addSeparator();
        jmenu1.add(jbutton);
        jmenu1.addSeparator();
        jmenu1.add(jmenuitem3);

        JMenu jmenu2 = new JMenu("Edición");
        JMenuItem jmenuitem4 = new JMenuItem("Cortar"),
            jmenuitem5 = new JMenuItem("Copiar"),
```

```
            jmenuitem6 = new JMenuItem("Pegar");

        jmenu2.add(jmenuitem4);
        jmenu2.add(jmenuitem5);
        jmenu2.add(jmenuitem6);

        jmenubar.add(jmenu1);
        jmenubar.add(jmenu2);

        setJMenuBar(jmenubar);
    }

    public void actionPerformed(ActionEvent e)
    {
       JButton jbutton = (JButton)e.getSource();
       showStatus(jbutton.getActionCommand());
    }
}
```

La figura 16.9 muestra el resultado de este código. Como vemos, se añade un botón completamente funcional al menú **Archivo** del applet. Este ejemplo se encuentra en el archivo menucontrols.java del CD.

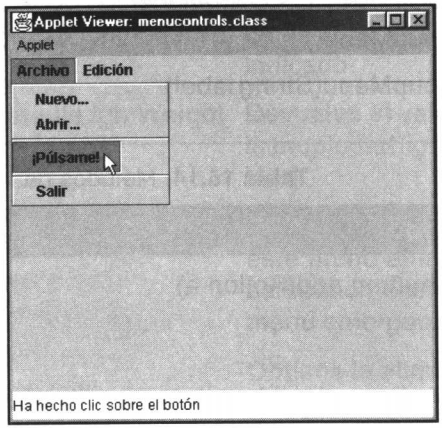


Figura 16.9. Añadir un botón a un menú.

Crear menús emergentes

El gran jefe está enfadado y dice, "¡Todas las nuevas características que mantener, todos estos programadores que contratar! ¡Qué coste! ¡Ahora son menús emergentes, nada menos!". "Está bien", decimos, "puedo añadirlos a nuestros programas si planifica un poco más de tiempo". El GJ grita: "¡El coste! ¡El coste!".

Los menús emergentes son esos menús que el usuario puede visualizar haciendo clic con el botón derecho del ratón cuando está situado sobre un componente. En la Swing, los menús emergentes quedan soportados por la clase JPopupMenu (lo adivinó). Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPopupMenu
```

La tabla 16.13 muestra los constructores para la clase JPopupMenu y la tabla 16.14 sus métodos.

Tabla 16.13. Constructores de la clase JPopupMenu.

Constructor	Descripción
JPopupMenu()	Construye un JPopupMenu.
JPopupMenu(String label)	Construye un JPopupMenu con el título indicado.

Tabla 16.14. Métodos de la clase JPopupMenu.

Método	Descripción
JMenuItem add(Action a)	Añade un nuevo elemento de menú al final del menú, que inicia la acción indicada.
JMenuItem add(JMenuItem menuItem)	Añade el elemento de menú indicado.
JMenuItem add(String s)	Construye un nuevo elemento con el texto indicado y lo añade al final del menú.
void addPopupMenuListener(PopupMenuListener l)	Añade un receptor PopupMenuListener.
void addSeparator()	Añade un nuevo separador al final de un menú.
protected PropertyChangeListener createActionChangeListener(JMenuItem b)	Crea un receptor de cambios de acción.

Método	Descripción
protected void firePopupMenuCanceled()	Informa a los receptores de este menú emergente que ha sido cancelado.
protected void firePopupMenuWillBecomeInvisible()	Informa a los PopupMenuListeners de que este menú emergente va a pasar a ser visible.
protected void firePopupMenuWillBecomeVisible()	Informa a los PopupMenuListeners de que este menú emergente será visible.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
Component getComponent()	Obtiene el Component utilizado para pintar el elemento recibido.
Component getComponentAtIndex (int i)	Obtiene el componente en el índice indicado.
int getComponentIndex(Component c)	Obtiene el índice del componente indicado.
static boolean getDefaultLightWeight PopupEnabled()	Devuelve el valor por defecto para la propiedad lightWeightPopupEnabled.
Component getInvoker()	Obtiene el componente que "invoca" este menú emergente (es decir, el componente en que se muestra este menú emergente).
String getLabel()	Obtiene la etiqueta del menú emergente.
Insets getMargin()	Obtiene el margen entre el borde del menú emergente y sus componentes contenidos.
SingleSelectionModel getSelectionModel()	Obtiene el modelo de objeto que procesa selecciones simples.
MenuElement[] getSubElements()	Debería devolver una matriz que contiene los subelementos.
PopupMenuUI getUI()	Obtiene el objeto apariencia que renderiza este componente.

Método	Descripción
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza este componente.
void insert(Action a, int index)	Inserta un elemento de menú para el objeto Action indicado en una posición dada.
void insert(Component component, int index)	Inserta el componente indicado en un menú en una posición dada.
boolean isBorderPainted()	Comprueba si el borde debería pintarse.
boolean isLightWeightPopupEnabled()	Devuelve true si los emergentes de poco peso se utilizan y false si se utilizan en cambio componentes de más peso.
boolean isVisible()	Devuelve true si el menú emergente está visible.
void menuSelectionChanged(boolean isIncluded)	Se llama cuando cambia la selección de menú.
void pack()	Distribuye el contenedor para que utilice el espacio mínimo para visualizar sus contenidos.
protected void paintBorder(Graphics g)	Pinta el borde del menú emergente.
protected String paramString()	Obtiene una representación de cadena de este JPopupMenu.
void processKeyEvent(KeyEvent e, MenuElement[] path, MenuSelection Manager manager)	Procesa un evento de tecla.
void processMouseEvent(Mouse Event event, MenuElement[] path, MenuSelectionManager manager)	Procesa un evento de ratón.
void remove(Component comp)	Elimina el componente indicado.
void remove(int pos)	Elimina el componente en la posición indicada.
void removePopupMenuListener (PopupMenuListener l)	Elimina un receptor PopupMenuListener.

Método	Descripción
void setBorderPainted(boolean b)	Asigna si el borde debería pintarse.
static void setDefaultLightWeightPopupEnabled(boolean aFlag)	Asigna el valor predeterminado para la propiedad lightWeightPopupEnabled.
void setInvoker(Component invoker)	Asigna el invocador para este menú emergente.
void setLabel(String label)	Asigna la etiqueta de menú emergente.
void setLightWeightPopupEnabled (boolean aFlag)	Selecciona si se utiliza un emergente de poco peso si es conveniente.
void setLocation(int x, int y)	Asigna la posición de la esquina superior izquierda del menú emergente.
void setPopupSize(Dimension d)	Asigna el tamaño del emergente utilizando el objeto Dimension.
void setPopupSize(int width, int height)	Asigna el tamaño del emergente.
void setSelected(Component sel)	Asigna el componente seleccionado en curso.
void setSelectionModel(SingleSelectionModel model)	Asigna al modelo de objeto para procesar selecciones simples.
void setUI(PopupMenuUI ui)	Asigna el objeto apariencia que dibuja este componente.
void setVisible(boolean b)	Asigna la visibilidad de menú.
void show(Component invoker, int x, int y)	Muestra un menú emergente.
void updateUI()	Llamada por UIFactory cuando cambia la apariencia.

Puede crear menús emergentes, añadirles nuevos elementos con el método add y después visualizarlos con el método show.

Vamos a examinar un ejemplo. En este caso, creamos un menú emergente con tres elementos (**Cortar**, **Copiar** y **Pegar**) que aparecerán cuando el usuario hace clic sobre el botón derecho del ratón. Los menús emergentes tienen que ser hijos de algún otro componente, por lo que añadiremos una

etiqueta a un applet y cubriremos el applet con la etiqueta. Aquí tiene la creación de la etiqueta y el menú emergente en el código:

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

    /*
 <APPLET
    CODE = popup.class
    WIDTH = 350
    HEIGHT = 280 >
 </APPLET>
    */

public class popup extends JApplet implements MouseListener
{
     JLabel jlabel = new JLabel("¡Haz clic con el botón derecho!",
JLabel.CENTER);
    JPopupMenu jpopupmenu = new JPopupMenu();

    public void init()
    {
        Container contentPane = getContentPane();

    jpopupmenu.add(new JMenuItem("Cortar", new ImageIcon("item.jpg")));
    jpopupmenu.add(new JMenuItem("Copiar", new ImageIcon("item.jpg")));
     jpopupmenu.add(new JMenuItem("Pegar", new ImageIcon("item.jpg")));

        jlabel.addMouseListener(this);
        contentPane.add(jlabel);
    }
    ...
}
```

Ahora, cuando el usuario hace clic sobre un botón de ratón, puede comprobar si se hizo sobre el botón derecho del ratón utilizando el método getModifiers de la clase MouseEvent y la máscara para el botón derecho del ratón (InputEvent.BUTTON3_MASK). Si se hizo clic sobre el botón derecho del ratón, mostraremos el nuevo menú emergente en la posición del ratón, como sigue:

```
public void mousePressed (MouseEvent e)
    {
        if((e.getModifiers() & InputEvent.BUTTON3_MASK) ==
            InputEvent.BUTTON3_MASK)
        jpopupmenu.show(jlabel, e.getX(), e.getY());
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
```

La figura 16.10 muestra el resultado. Cuando el usuario hace clic con el botón derecho sobre la etiqueta en el applet, el menú emergente aparece en la posición del ratón, como se muestran en la figura. Eso es lo único necesario para visualizar menús emergentes. Este ejemplo se encuentra en el archivo popup.java del CD.

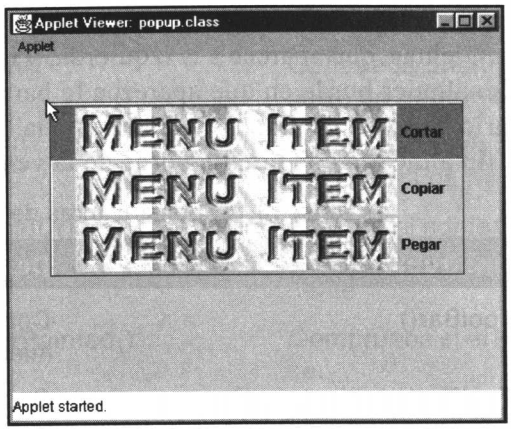


Figura 16.10. Visualizar un menú emergente.

Crear barras de herramientas

"Los menús están bien", dice el programador novato, "pero algunas veces trabajo demasiado rápido para usarlos, por lo que tengo muchos aceleradores de menús en mi programa. Pero ahora quiero olvidarlos". Sonreímos y decimos: "¿Ha probado a añadir una barra de herramientas?"

Las barras de herramientas visualizan botones y otros controles que representan acciones comunes de su programa, como guardar un archivo o pegar los contenidos del portapapeles. Los botones de la barra de herramientas a veces representan elementos de menús usados con frecuencia en su sistema de menús. En la Swing, utilizamos la clase JToolBar para crear barras de herramientas. Aquí tiene el diagrama de herencia de la clase JToolBar:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JToolBar
```

La tabla 16.15 muestra los constructores de la clase JToolBar y la tabla 16.16 sus métodos.

Aquí tiene un ejemplo en el que añadimos una barra de herramientas con los botones a un programa. Observe que, en cierta forma, una barra de herramientas actúa como un componente que contiene otros componentes; tiene que añadirlo a la distribución de su programa donde quiera. Sin embargo, cuando lo ha hecho, el usuario puede arrastrar el indicador de la barra de herramientas, que aparece a la izquierda, y realinear la barra de herramientas con cualquier borde en que aparezca la barra de herramientas. De hecho, el usuario puede dejar flotar sencillamente la barra de herramientas en el espacio, sin alinearla con ningún borde de la ventana.

Tabla 16.15. Constructores de la clase JToolBar.

Constructor	Descripción
JToolBar()	Construye una nueva barra de herramientas.
JToolBar(int orientation)	Construye una nueva barra de herramientas con la orientación dada.

Tabla 16.16. Métodos de la clase JToolBar.

Método	Descripción
JButton add(Action a)	Añade un nuevo JButton que inicia la acción.
protected void addImpl(Component comp, Object constraints, int index)	Añade el componente indicado a este contenedor en la posición indicada.
void addSeparator()	Añade un separador de barra de herramientas.
void addSeparator(Dimension size)	Añade un separador de barra de herramientas con las dimensiones dadas.
protected PropertyChangeListener createActionChangeListener(JButtonb)	Crea un receptor de cambio de acción.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
Component getComponentAtIndex (int i)	Obtiene el componente en el índice indicado.

Método	Descripción
int getComponentIndex(Component c)	Obtiene el índice del componente indicado.
Insets getMargin()	Obtiene el margen entre el borde de la barra de herramientas y sus botones.
int getOrientation()	Obtiene la orientación actual de la barra de herramientas.
ToolBarUI getUI()	Obtiene la interfaz de usuario actual de la barra de herramientas.
String getUIClassID()	Obtiene el nombre de la clase apariencia que renderiza este componente.
boolean isBorderPainted()	Comprueba si el borde debe pintarse.
boolean isFloatable()	Devuelve true si la barra de herramientas puede ser arrastrada por el usuario.
protected void paintBorder(Graphics g)	Quita el borde de la barra de herramientas.
protected String paramString()	Obtiene una representación de cadena de esta barra de herramientas.
void remove(Component comp)	Elimina el componente de la barra de herramientas.
void setBorderPainted(boolean b)	Asigna si el borde debería pintarse.
void setFloatable(boolean b)	Asigna si la barra de herramientas se puede hacer flotar.
void setMargin(Insets m)	Asigna el margen del borde de la barra de herramientas y sus botones.
void setOrientation(int o)	Asigna la orientación de la barra de herramientas.
void setUI(ToolBarUI ui)	Asigna el objeto apariencia que renderiza este componente.
void updateUI()	Llamada por UIFactory cuando cambia la apariencia.

En el código de este ejemplo, utilizamos sencillamente el método add de la clase JToolBar para añadir dos botones a una barra de herramientas y utilizamos el método addSeparator para añadir algo de espacio entre los botones. También añadimos código para hacer que el applet visualice los botones de la barra de herramientas sobre los que ha hecho clic el usuario y añadimos la barra de herramientas a la sección norte del panel de contenidos de la distribución de borde. Aquí tiene el código:

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

    /*
 <APPLET
    CODE = toolbar.class
    WIDTH = 500
    HEIGHT = 280 >
 </APPLET>
    */

public class toolbar extends JApplet implements ActionListener, ItemListener
{
  JButton jbutton1 = new JButton("Botón 1", new ImageIcon("button.jpg"));
  JButton jbutton2 = new JButton("Botón 2", new ImageIcon("button.jpg"));

    public void init()
    {
        Container contentPane = getContentPane();

        JToolBar jtoolbar = new JToolBar();

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);

        jtoolbar.add(jbutton1);
        jtoolbar.addSeparator();
        jtoolbar.add(jbutton2);

        contentPane.add(jtoolbar, BorderLayout.NORTH);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            showStatus("Hizo clic sobre el botón 1");
        } else if (e.getSource() == jbutton2) {
            showStatus("Hizo clic sobre el botón 2");
        }
    }

}
```

La figura 16.11 muestra el resultado de este código, donde puede ver los dos botones de la barra de herramientas. Observe también el indicador de la barra de herramientas a la izquierda de la misma, que el usuario puede utilizar para moverla, alineándola como sea necesario o dejándola flotar libremente. Este ejemplo (el archivo toolbar.java del CD) es un éxito.

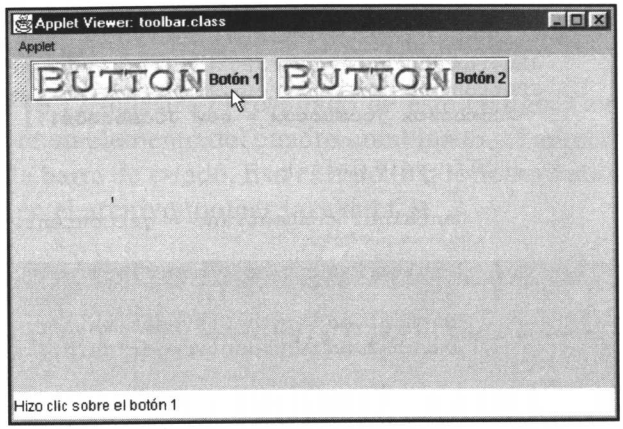


Figura 16.11. Mostrar una barra de herramientas con dos botones.

Añadir cuadros combinados y otros controles a barras de herramientas

El especialista de soporte a productos está poco feliz y dice, "Los usuarios se están quejando de nuevo. Tenemos veinte botones en la barra de herramientas para seleccionar el tipo de letra y no existe espacio para otros botones". "Mmh", decimos, "eso podría ser un problema. Pondremos las diversas opciones de tipo de letra en un cuadro combinado en vez de en la barra de herramientas". El habitualmente lúgubre ESP se anima momentáneamente.

Puede añadir cuadros combinados y otros controles a barras de herramientas utilizando el método add de la clase JToolBar. Aquí tiene un ejemplo en el que añadiremos un cuadro combinado al ejemplo del cuadro de herramientas desarrollado en el tema previo:

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

```
    /*
 <APPLET
    CODE = toolbar.class
    WIDTH = 500
    HEIGHT = 280 >
 </APPLET>
    */

public class toolbar extends JApplet implements ActionListener, ItemListener
{
  JButton jbutton1 = new JButton("Botón 1", new ImageIcon("button.jpg"));
  JButton jbutton2 = new JButton("Botón 2", new ImageIcon("button.jpg"));
    JComboBox jcombobox = new JComboBox();

    public void init()
    {
        Container contentPane = getContentPane();

        JToolBar jtoolbar = new JToolBar();

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);

        jcombobox.addItem("Elemento 1");
        jcombobox.addItem("Elemento 2");
        jcombobox.addItem("Elemento 3");
        jcombobox.addItem("Elemento 4");
        jcombobox.addItemListener(this);

        jtoolbar.add(jbutton1);
        jtoolbar.addSeparator();
        jtoolbar.add(jbutton2);
        jtoolbar.addSeparator();
        jtoolbar.add(jcombobox);

        contentPane.add(jtoolbar, BorderLayout.NORTH);
    }

   ...
}
```

Cuando el usuario selecciona un elemento del cuadro combinado, visualizamos ese elemento en la barra de estado:

```
public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            showStatus("Hizo clic sobre el botón 1");
        } else if (e.getSource() == jbutton2) {
            showStatus("Hizo clic sobre el botón 2");
        }
    }

    public void itemStateChanged(ItemEvent e)
```

```
{
    String outString = "";

    if(e.getStateChange() == ItemEvent.SELECTED)
        outString += "Seleccionado: " + (String)e.getItem();
    else
        outString += "Deseleccionado: " + (String)e.getItem();

    showStatus(outString);
}
```

La figura 16.12 muestra el resultado de este código. Cuando el usuario hace clic sobre un elemento del cuadro combinado, el applet visualizará ese elemento en la barra de estado. Eso es todo lo que necesitamos. Este ejemplo se encuentra en el archivo toolbar.java del CD.

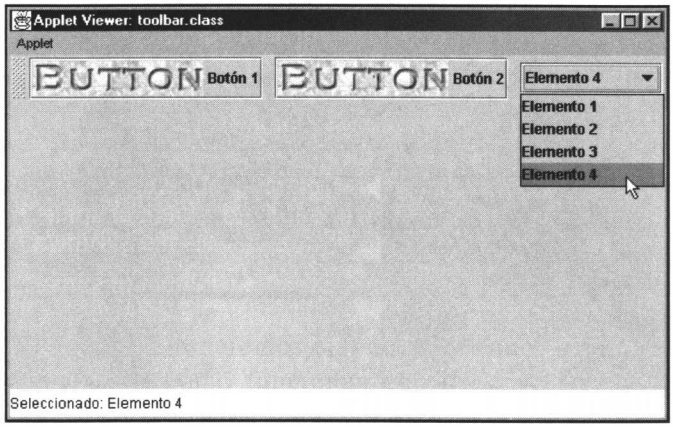


Figura 16.12. Añadir un cuadro combinado a una barra de herramientas.

17 Swing: ventanas, paneles, marcos internos y cuadros de diálogo

En este capítulo, examinaremos el manejo de todo tipo de ventanas en la Swing. Aprenderemos cómo funcionan con las clases JWindow, JFrame, JDesktopPane, JInternalFrame, JOptionPane y JDialog. Estas clases pueden dividirse convenientemente en dos categorías: ventanas y cuadros de diálogo.

Ventanas

Las clases JWindow y JFrame son el equivalente de las clases Window y Frame en AWT y sirven para el mismo propósito. Puede utilizar la clase JWindow para crear una ventana simple; de hecho, no es más que un rectángulo blanco. Aunque puede añadir bordes y controles a los objetos JWindow, normalmente utilizará la clase JFrame para crear ventanas que presente al usuario.

Las clases JDesktopPane y JInternalFrame son nuevas en Swing, aunque juntas representan algo que ha llegado a ser muy común en las IU: una interfaz de documentos múltiple. Los objetos de la clase JDesktopPane presentan un espacio en el que puede visualizar múltiples marcos internos de la

clase JInternalFrame. Por ejemplo, una aplicación de procesamiento de textos podría permitir al usuario abrir varias vistas del mismo documento, o de múltiples documentos, utilizando un escritorio con varias ventanas de marcos internos. Los marcos internos son de poco peso, y dibujan ventanas que aparecen dentro de otras ventanas. De hecho, los paneles de escritorio también son componentes de poco peso, derivados de JLayeredPane. Puede añadir ventanas de marcos internos a las capas seleccionadas de un panel de escritorio.

Cuadros de diálogo

La Swing proporciona bastante soporte para cuadros de diálogo con la clase JOptionPane. Mediante el uso de métodos de esta clase, puede visualizar toda clase de cuadros de diálogo (cuadros de diálogo de mensaje, cuadros de diálogo de confirmación, cuadros de diálogo de entrada y muchos más). Como verá aquí, es sencillo crear cuadros de diálogo utilizando JOptionPane.

Además de JOptionPane, la Swing también posee la clase JDialog, que puede utilizar como base para sus propias clases de cuadro de diálogo personalizadas. Veremos cómo utilizar JOptionPane y JDialog en profundidad en este capítulo.

Es suficiente como resumen de lo que aparece en este capítulo. Como puede ver, habrá mucho más.

Crear una ventana

El programador novato aparece y dice, "Quiero crear una ventana completamente personalizada en el programa". "Para eso", decimos, "necesita la clase JWindow, que visualiza una ventana completamente limpia". "Bien", dice el PN, "¿puedes escribir el código para mí?"

La clase JWindow es muy similar a la clase Window del AWT, debido a que visualiza una ventana vacía que puede personalizar como desee. De hecho, los objetos JWindow tienen tan pocos adornos que generalmente utilizará en su lugar la clase JFrame. Sin embargo, si de verdad necesita simplicidad, JWindow es el punto de inicio. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +--java.awt.Component
```

```
|
+-java.awt.Container
        |
        +-java.awt.Window
                |
                +-javax.swing.JWindow
```

La tabla 17.1 muestra los constructores para JWindow y la tabla 17.2 sus métodos.

Tabla 17.1. Constructores de la clase JWindow.

Constructor	Descripción
JWindow()	Construye una ventana.
JWindow(Frame owner)	Construye una ventana con el marco propietario indicado.
JWindow(Window owner)	Construye una ventana con la ventana propietaria indicada.

Tabla 17.2. Métodos de la clase JWindow.

Método	Descripción
protected void addImpl(Component comp, Object constraints, int index)	Añade un componente al panel de contenidos en su lugar.
protected JRootPane createRootPane()	Llamada por los métodos constructores para crear el panel raíz predeterminado.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
Container getContentPane()	Obtiene el objeto contentPane para esta ventana.
Component getGlassPane()	Obtiene el objeto glassPane para esta ventana.
JLayeredPane getLayeredPane()	Obtiene el objeto layeredPane para esta ventana.
JRootPane getRootPane()	Obtiene el objeto rootPane para esta ventana.
protected boolean isRootPaneCheckingEnabled()	Determina si las llamadas a add y setLayout producen una excepción que debe lanzarse.

Método	Descripción
protected String paramString()	Obtiene una representación de cadena de este JWindow.
void remove (Component comp)	Elimina el componente indicado de este contenedor.
void setContentPane(Container contentPane)	Asigna la propiedad contentPane.
void setGlassPane(Component glassPane)	Asigna la propiedad glassPane.
void setLayeredPane(JLayeredPane layeredPane)	Asigna la propiedad layeredPane.
void setLayout(LayoutManager manager)	Por defecto, la distribución de este componente no debe asignarse.
protected void setRootPane(JRootPane root)	Asigna la propiedad rootPane.
protected void setRootPaneCheckingEnabled(boolean enabled)	Determina si las llamadas a add y setLayout pueden producir una excepción.
protected void windowInit()	Llamada por los constructores para inicializar la propiedad JWindow.

Cuando crea un objeto JWindow, puede añadirle un borde con el método setBorder del panel raíz, mostrar la ventana con el método show, ocultarla con el método hide y eliminarla con el método dispose. Observe que la distribución por defecto de JWindow es la distribución de borde.

Crearemos un objeto JWindow ahora en un ejemplo. En este caso, añadiremos un botón a un applet y visualizaremos la ventana cuando se pulse el botón. También añadiremos un borde a la ventana y un botón que, cuando se pulse, eliminará la ventana. Observe que los objetos JWindow tienen paneles de contenido como cualquier otra ventana, por lo que añadimos un botón a ese panel. Observe también que antes de que se pueda mostrar una ventana, debe tener un tamaño, que asignamos con el método setBounds. Aquí tiene el código:

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

    /*
<APPLET
```

```
    CODE = window.class
    WIDTH = 350
    HEIGHT = 280 >

    */

public class window extends JApplet implements ActionListener
{
    JWindow jwindow = new JWindow();

    public void init()
    {
        Container contentPane = getContentPane();
        JButton jbutton = new JButton("Mostrar ventana");
        JButton jwindowbutton = new JButton("Cerrar");

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jbutton);

        jwindow.getRootPane().setBorder(
            BorderFactory.createRaisedBevelBorder()
        );

        Container windowContentPane = jwindow.getContentPane();
        windowContentPane.setLayout(new FlowLayout());
        windowContentPane.add(jwindowbutton);

        jwindowbutton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                jwindow.dispose();
            }
        });

        jbutton.addActionListener(this);

    }

    public void actionPerformed(ActionEvent e)
    {
        jwindow.setBounds(200, 200, 100, 100);

        jwindow.show();
    }
}
```

La figura 17.1 muestra el resultado de este código, donde puede ver la nueva ventana que aparece cuando el usuario hace clic sobre el botón **Mostrar ventana**. Como vemos, la ventana tiene un borde y un botón (así como un aviso que añade la Swing que indica que la ventana es una ventana de applet). Eso es todo. Este ejemplo se encuentra en el archivo window.java del CD.



Truco: Aunque en este ejemplo pasamos valores numéricos al método setBounds de la ventana, también puede asignar la posición de la ventana con respecto a su ventana padre. Primero, obtenga la localización del panel de contenido de la ventana del applet como sigue: **Point loc = contentPane.getLocation()**. Este punto está en coordenadas locales, por lo que debe convertirlo para utilizar coordenadas de pantalla, como sigue: **SwingUtilities.convertPointToScreen(loc, contentPane)**. A partir de este momento es libre de utilizar **loc** con **setBounds**.

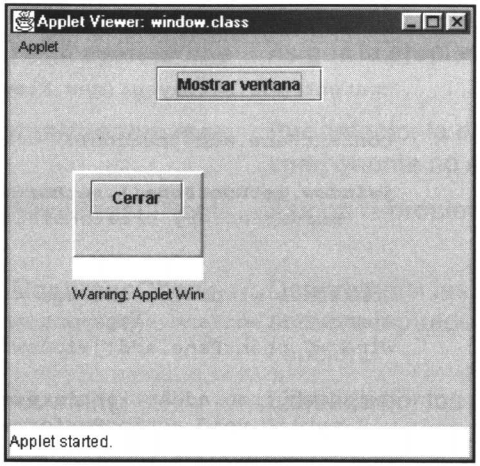


Figura 17.1. Una ventana JWindow básica.

Crear una ventana de marco

"Bien", dice el programador novato, "estaba equivocado. No quiero comenzar sólo con una ventana básica; quiero utilizar la ventana de marco que ya soporte el cambio de tamaño, minimizar, maximizar y todo eso". "Bien", decimos. "Parece que quiere una clase JFrame".

Hemos utilizado la clase JFrame como base de las aplicaciones Swing, por lo que ya está familiarizado con ella.

Vamos a incluirla en este capítulo sobre ventanas y cuadros de diálogo para completar. En el capítulo 11, encontrará los campos de la clase JFrame que se muestran en la tabla 11.15, la tabla 11.16 muestra sus constructores y la tabla 11.17 sus métodos.

Aquí tiene un ejemplo rápido que utiliza una ventana JFrame en la que añadimos un panel y después dibujamos y añadimos un campo de texto en el

panel. Debido a que las ventanas JFrame se utilizan como base de las aplicaciones Swing, haremos una aplicación con este ejemplo. Aquí tiene el código:

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

    /*
<APPLET
    CODE = jframe.class
    WIDTH = 350
    HEIGHT = 280>

    */

public class jframe extends JFrame
{
    jpanel j;

    public jframe()
    {
        super("Aplicación Swing");

        Container contentPane = getContentPane();
        j = new jpanel();
        contentPane.add(j);
    }

    public static void main(String args[])
    {
        final JFrame f = new jframe();

        f.setBounds(100, 100, 300, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

class jpanel extends JPanel
{
    JTextField jtextfield = new JTextField("¡Hola desde Swing!");

    jpanel()
    {
        setBackground(Color.white);
        add(jtextfield);
    }
```

```
    public void paintComponent (Graphics g)
    {
        super.paintComponent(g);
        g.drawString("¡Hola desde Swing!", 0, 60);
    }
}
```

La figura 17.2 muestra el resultado de este código. Como puede ver en la figura, la ventana de marco ya muestra los botones de minimizar, maximizar y cerrar. Este ejemplo se encuentra en el archivo jframe.java del CD. Para saber más sobre el uso de JFrame, como por ejemplo los detalles para cerrar estas ventanas, consulte el capítulo 11.



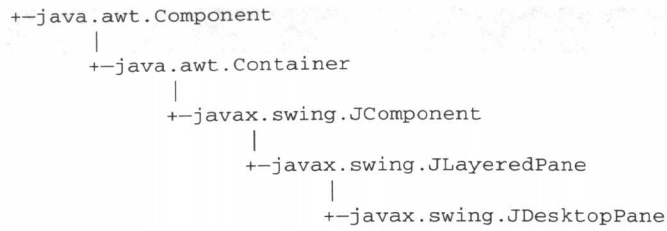
Figura 17.2. Uso de la clase JFrame.

Crear un panel de escritorio

El gran jefe aparece y dice, "Nuestro procesador de textos Java no es competitivo. La competencia puede abrir cinco documentos al mismo tiempo". "Eso no es nada", decimos. "Usando un panel de escritorio, puede abrir diez documentos a la vez y visualizarlos todos". "Estás contratado", dice el GJ. Decimos, "ya trabajo aquí".

El panel de escritorio de Swing representa un escritorio y nos permite visualizar múltiples objetos. En el siguiente tema, añadiremos ventanas de marcos internas a un panel de escritorio. Aquí tiene el diagrama de herencia para la clase JDesktopPane, que es la clase que utiliza la Swing para soportar paneles de escritorio:

```
java.lang.Object
  |
```



La tabla 17.3 muestra los constructores de la clase JDesktopPane y la tabla 17.4 muestra sus métodos.

Tabla 17.3. El constructor de la clase JDesktopPane.

Constructor	Descripción
JDesktopPane()	Construye un nuevo JDesktopPane.

Tabla 17.4. Métodos de la clase JDesktopPane.

Método	Descripción
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
JInternalFrame[] getAllFrames()	Obtiene todos los JInternalFrames mostrados en el escritorio.
JInternalFrame[] getAllFramesInLayer(int layer)	Obtiene todos los JInternalFrames mostrados en una capa del escritorio.
DesktopManager getDesktopManager()	Obtiene el DesktopManger que maneja la IU específica de escritorio.
DesktopPaneUI getUI()	Obtiene el objeto apariencia que dibuja este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que dibujará el componente.
boolean isOpaque()	Devuelve true si este componente pinta todos los puntos en su rango.
protected String paramString()	Obtiene la representación de cadena de este JDesktopPane.

Método	Descripción
void setDesktopManager(DesktopManager d)	Obtiene el DesktopManger que manejará las acciones muy específicas del escritorio.
void setUI(DesktopPaneUI ui)	Obtiene el objeto apariencia que dibujará el componente.
void updateUI()	Llamada por UIManager cuando cambia la apariencia.

Aquí tiene un ejemplo en el que únicamente añadimos un panel de escritorio al panel de contenidos de un applet. Éste es el código:

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

    /*
<APPLET
    CODE = desktopframe.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
    */

public class desktopframe extends JApplet
{
    JDesktopPane jdesktoppane = new JDesktopPane();

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.add(jdesktoppane, BorderLayout.CENTER);
    }
}
```

La figura 17.3 muestra el resultado de este código. Vemos en ella un panel de escritorio vacío, pero no es gran cosa; los paneles de escritorio son útiles como contenedores, no como componentes, por lo que necesitamos añadirles algo. Lo haremos en el siguiente tema, donde veremos los marcos internos.

Crear marcos internos

El gran jefe no está impresionado con el panel de escritorio de nuestro programa. "¿Cuál es la clave?", pregunta el GJ. "La clave", decimos, "es que

puede visualizar ventanas dentro del panel de escritorio". "Convénzame", dice el jefe. "No hay problema", contestamos.

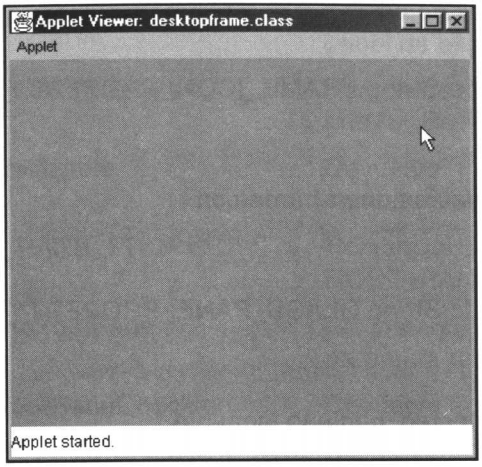


Figura 17.3. Un panel de escritorio básico.

Los marcos internos, soportados por la clase JInternalFrame, son ventanas de marco de poco peso que aparecen dentro de otras ventanas. Esta es una de las clases más grandes y populares de la Swing y la haremos funcionar aquí. Veamos el diagrama de herencia para la clase JInternalFrame:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JInternalFrame
```

La tabla 17.5 muestra los campos de la clase JInternalFrame, la tabla 17.6 sus constructores y la tabla 17.7 sus métodos.

Tabla 17.5. Campos de la clase JInternalFrame.

Campo	Descripción
protected boolean closable	Indica si el marco se puede cerrar.
static String CONTENT_PANE_PROPERTY	Nombre de la propiedad ligada.

Campo	Descripción
protected JInternalFrame.JDesktopIcon desktopIcon	El icono visualizado cuando el marco se convierte en un icono.
static String FRAME_ICON_PROPERTY	El nombre de la propiedad ligada FRAME_ICON_PROPERTY.
protected Icon frameIcon	El icono mostrado en la esquina superior izquierda del marco.
static String GLASS_PANE_PROPERTY	El nombre de la propiedad ligada GLASS_PANE_PROPERTY.
protected boolean iconable	Devuelve true si el marco se puede minimizar y visualizar con una imagen de icono.
static String IS_CLOSED_PROPERTY	Propiedad que indica que el marco se puede cerrar.
static String IS_ICON_PROPERTY	Propiedad que indica que el marco se ha convertido en un icono.
static String IS_MAXIMUM_PROPERTY	Propiedad que indica que el marco está maximizado.
static String IS_SELECTED_PROPERTY	Propiedad que indica que este marco tiene el estado de selección.
protected boolean isClosed	Devuelve true si el marco ha sido cerrado.
protected boolean isIcon	Devuelve true si el marco ha sido convertido en un icono.
protected boolean isMaximum	Devuelve true si el marco se ha expandido a su tamaño máximo.
protected boolean isSelected	Devuelve true si el marco está seleccionado en la actualidad.
static String LAYERED_PANE_PROPERTY	Nombre de la propiedad ligada LAYERED_PANE_PROPERTY.

Campo	Descripción
protected boolean maximizable	Devuelve true si el marco se puede expandir al tamaño del panel de escritorio.
static String MENU_BAR_PROPERTY	Nombre de la propiedad ligada MENU_BAR_PROPERTY.
protected boolean resizable	Devuelve true si se puede cambiar el tamaño del marco.
static String ROOT_PANE_PROPERTY	Nombre de la propiedad ligada ROOT_PANE_PROPERTY.
protected JRootPane rootPane	La instancia de JRootPane para este marco.
protected boolean rootPaneChecking-Enabled	Devuelve true cuando las llamadas a add y setLayout pueden generar una excepción.
protected String title	El título a visualizar en la barra de título del marco.
static String TITLE_PROPERTY	Nombre de la propiedad ligada TITLE_PROPERTY.

Tabla 17.6. Constructores de la clase JInternalFrame.

Constructor	Descripción
JInternalFrame()	Construye un JInternalFrame que no se puede convertir en icono, no se puede cambiar de tamaño, no se puede cerrar y no puede expandirse hasta su tamaño máximo.
JInternalFrame(String title)	Construye un JInternalFrame que no se puede convertir en icono, no se puede cambiar de tamaño, no se puede cerrar y no puede expandirse hasta su tamaño máximo con un texto dado.
JInternalFrame(String title, boolean resizable)	Construye un JInternalFrame que no se puede convertir en icono, no se puede cerrar y no

Constructor	Descripción
	puede expandirse hasta su tamaño máximo, pero que puede cambiar su tamaño si resizeable es true.
JInternalFrame(String title, boolean resizable, boolean closable)	Construye un JInternalFrame que no se puede convertir en icono, y no puede expandirse hasta su tamaño máximo, pero que puede cambiar su tamaño, y cerrarse con un texto dado.
JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)	Construye un JInternalFrame que no se puede convertir en icono, con el texto indicado, que puede cambiar de tamaño, cerrarse y expandirse hasta su tamaño máximo según se indique.
JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)	Construye un JInternalFrame con el texto indicado y con la posibilidad de cambio de tamaño, cerrarse, expandirse hasta su tamaño máximo y convertirse en un icono, según se indique.

Tabla 17.7. Métodos de la clase JInternalFrame.

Método	Descripción
protected void addImpl(Component comp, Object constraints, int index)	Añade un componente hijo a contentPane.
void addInternalFrameListener(InternalFrameListener l)	Añade un receptor de marco interno dado para recibir los eventos de marco interno a partir de éste.
protected JRootPane createRootPane()	Llamada por el constructor para configurar el JRootPane.
void dispose()	Elimina este marco interno.
protected void fireInternalFrameEvent (int id)	Dispara un evento de marco interno.

Método	Descripción
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
Container getContentPane()	Obtiene el contenPane.
int getDefaultCloseOperation()	Obtiene la operación predeterminada que ocurre cuando el usuario cierra esta ventana.
JInternalFrame.JDesktopIcon getDesktopIcon()	Obtiene el JDesktopIcon utilizado cuando se convierte en un icono este JInternalFrame.
JDesktopPane getDesktopPane()	Busca en la jerarquía una instancia de JDesktop.
Icon getFrameIcon()	Obtiene la imagen visualizada en la barra de título del marco.
Component getGlassPane()	Obtiene el objeto glassPane para este marco.
JMenuBar getJMenuBar()	Obtiene el JMenuBar actual para este JInternalFrame o nulo si la barra de menú no se asignó.
int getLayer()	Un método conveniente para obtener el atributo capa de este componente.
JLayeredPane getLayeredPane()	Obtiene el objeto layeredPane para este marco.
JMenuBar getMenuBar()	Obsoleto. Reemplazado por getJMenuBar().
JRootPane getRootPane()	Obtiene el objeto rootPane para este marco.
String getTitle()	Obtiene el título del JInternalFrame.
InternalFrameUI getUI()	Obtiene el objeto apariencia que dibuja este JInternalFrame.
String getUIClassID()	Obtiene el nombre de la clase apariencia que dibuja este componente.

Método	Descripción
String getWarningString()	Obtiene la cadena de aviso que se visualiza con esta ventana.
boolean isClosable()	Obtiene si se puede cerrar por alguna acción de usuario este JInternalFrame.
boolean isClosed()	Devuelve true si el JInternal-Frame está actualmente cerrado.
boolean isIcon()	Devuelve true si el JInternal-Frame está actualmente convertido en un icono.
boolean isIconifiable()	Devuelve true si el JInternal-Frame se puede convertir en un icono por alguna acción del usuario.
boolean isMaximizable()	Devuelve true si el JInternal-Frame puede maximizarse por alguna acción del usuario
boolean isMaximum()	Devuelve true si el JInternal-Frame está actualmente en su estado máximo.
boolean isResizable()	Devuelve true si el JInternal-Frame se puede cambiar de tamaño por una acción de usuario.
protected boolean isRootPaneChecking-Enabled()	Devuelve true si las llamadas a add y setLayout pueden generar una excepción.
boolean isSelected()	Devuelve true si el JInternal-Frame es el marco seleccionado actualmente.
void moveToBack()	Mueve este componente a la posición -1 si su padre es un JLayeredPane.
void moveToFront()	Mueve este componente a la posición 0 si su padre es un JLayeredPane.

Método	Descripción
void pack()	Distribuye los hijos de este JInternalFrame para usar su tamaño favorito.
protected void paintComponent (Graphics g)	Llama al método paint.
protected String paramString()	Obtiene una representación de cadena de este JInternal-Frame.
void remove (Component comp)	Elimina el componente indicado.
void removeInternalFrameListener(Internal-FrameListener l)	Elimina el receptor de marcos internos indicado.
void reshape(int x, int y, int width, int height)	Mueve o cambia de tamaño el componente.
void setClosable(boolean b)	Determina si este JInternal-Frame se puede cerrar por alguna acción de usuario.
void setClosed(boolean b)	Si pasamos true a este método cerramos el marco.
void setContentPane(Container c)	Asigna este panel de contenido del JInternalFrame.
void setDefaultCloseOperation(int operation)	Asigna la operación que ocurrirá por defecto cuando el usuario cierre esta ventana.
void setDesktopIcon(JInternalFrame. JDesktopIcon d)	Asigna el JDesktopIcon asociado con este JInternalFrame.
void setFrameIcon(Icon icon)	Asigna una imagen que se visualiza en la barra de título del marco (normalmente en la esquina superior izquierda).
void setGlassPane(Component glass)	Asigna la propiedad glassPane del JInternalFrame.
void setIcon(boolean b)	Convierte en un icono y realiza la operación inversa con el marco.
void setIconifiable(boolean b)	Especifica que JInternalFrame se puede convertir en un

Método	Descripción
	icono por alguna operación de usuario.
void setJMenuBar(JMenuBar m)	Asigna el JMenuBar de este JInternalFrame.
void setLayer(Integer layer)	Asigna el atributo capa de este componente.
void setLayeredPane(JLayeredPane layered)	Asigna la propiedad layeredPane de este JInternalFrame.
void setLayout(LayoutManager manager)	Asigna la distribución del JInternalFrame.
void setMaximizable(boolean b)	Especifica si el JInternalFrame puede expandirse hasta su tamaño máximo por alguna acción de usuario.
void setMaximum(boolean b)	Expande el marco a su tamaño máximo o lo restaura.
void setMenuBar(JMenuBar m)	Obsoleto. Reemplazado por setJMenuBar(JMenuBar m).
void setResizable(boolean b)	Especifica que el JInternalFrame se puede cambiar de tamaño por alguna acción de usuario.
protected void setRootPane(JRootPane root)	Asigna la propiedad rootPane.
protected void setRootPaneCheckingEnabled(boolean enabled)	Determina si las llamadas a add o setLayout pueden generar una excepción.
void setSelected(boolean selected)	Selecciona el JInternalFrame o lo deselecciona.
void setTitle(String title)	Asigna el título del JInternalFrame.
void setUI(InternalFrameUI ui)	Asigna el IU delegado por este JInternalFrame.
void setVisible(boolean b)	Asigna el estado visible del objeto.

Método	Descripción
void show()	Muestra este marco interno y lo pasa a primer plano.
void toBack()	Envía este marco interno al fondo.
void toFront()	Envía a primer plano este marco interno.
void updateUI()	Llamada por UIManager cuando cambia la apariencia.

Ahora adjuntaremos un ejemplo que muestra cómo utilizar marcos internos en un panel de escritorio. Comience añadiendo un panel de escritorio a un applet y añada también un objeto JPanel. Sitúe un botón con el título "nuevo marco interno" en el panel. Cuando el usuario hace clic sobre ese botón, visualiza un nuevo marco interno en el panel de escritorio. Cada marco interno contendrá un objeto área de texto de la clase JTextArea (que veremos en unos pocos capítulos; para los propósitos de este ejemplo, actuará como la clase JTextArea de la AWT), permitiendo al usuario introducir texto.

Aquí tiene la configuración del applet con un panel de escritorio y el botón **Nuevo marco interno**:

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

    /*
<APPLET
    CODE = internalframe.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET
    */

public class internalframe extends JApplet implements ActionListener
{
    JDesktopPane jdesktoppane = new JDesktopPane();

    public void init()
    {
        JPanel jpanel = new JPanel();
        Container contentPane = getContentPane();

        JButton jbutton = new JButton("Nuevo marco interno");

        jpanel.add(jbutton);
```

```
        contentPane.add(jpanel, BorderLayout.SOUTH);
        contentPane.add(jdesktoppane, BorderLayout.CENTER);

        jbutton.addActionListener(this);
    }
    ...
}
```

La acción real tiene lugar cuando el usuario hace clic sobre el botón para visualizar un nuevo marco interno. Proporcionamos a cada marco interno un nuevo texto en su barra de título, por lo que debemos controlar el número de marcos internos en una nueva variable:

```
public class internalframe extends JApplet implements ActionListener
{
    JDesktopPane jdesktoppane = new JDesktopPane();
    static int framenumber = 1;

    public void init()
    {
       ...
    }
```

A continuación, creamos, configuramos y añadimos nuevos marcos internos al panel de escritorio. Aquí tiene cómo crear y configurar los marcos internos, asignar su posición y título. Los habilitaremos para que el usuario pueda cerrarlos, cambiar su tamaño, maximizar y minimizar estos marcos y añadir un área de texto al panel de contenidos del marco. Observe, en particular, que los marcos internos deben hacerse visibles con el método setVisible. Una vez se añaden los componentes al panel de contenidos, el método pack se utiliza para cambiar el tamaño de la ventana de marco para que coincida con sus contenidos. Observe también que cuando se añaden marcos internos a paneles de escritorio utilizando el método add, debe especificarse una capa. Aquí tiene el código:

```
public void actionPerformed(ActionEvent event)
{
    JInternalFrame jinternalframe = new JInternalFrame();
    Container contentPane = jinternalframe.getContentPane();

    jinternalframe.setLocation(20, 20);
    jinternalframe.setTitle("Marco interno " + framenumber);
    framenumber++;

    jinternalframe.setClosable(true);
    jinternalframe.setResizable(true);
    jinternalframe.setMaximizable(true);
    jinternalframe.setIconifiable(true);

    jinternalframe.setVisible(true);
```

```
        contentPane.setLayout(new FlowLayout());
        contentPane.add(new JTextArea(5, 15), "Center");
        jinternalframe.pack();

        jdesktoppane.add(jinternalframe, 2);
    }
```

La figura 17.4 muestra el resultado de este código. Como verá, puede crear múltiples ventanas de marco interno en este applet y puede cambiarlas de tamaño y moverlas a voluntad. Incluso también puede minimizarlas, como muestra la figura. Este applet funciona tal y como fue diseñado y lo encontrará en el archivo internalframe.java del CD.

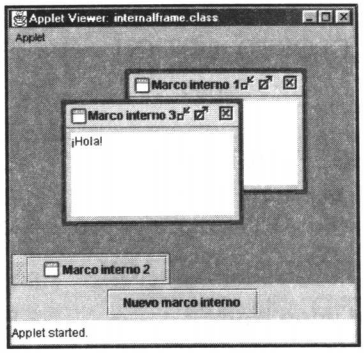


Figura 17.4. Creación de marcos internos en un panel de escritorio.

Uso de JOptionPane para crear cuadros de diálogo

El programador novato está como loco y dice, "El especialista de soporte a productos quiere que añada 12 nuevos cuadros de diálogo a mi programa. ¿No es codificar mucho?". "Probablemente no", decimos, "cuando utilice la clase JOptionPane de la Swing". "Oh", dice el PN, calmándose y comenzando a sonreír.

La clase JOptionPane soporta un panel que utilizamos en cuadros de diálogo; de hecho, también proporciona métodos convenientes para la creación de cuadros de diálogo que contienen paneles de opciones. Aquí tiene el diagrama de herencia para JOptionPane:

```
java.lang.Object
  |
  +-java.awt.Component
        |
```

```
+--java.awt.Container
      |
      +--javax.swing.JComponent
            |
            +--javax.swing.JOptionPane
```

La tabla 17.8 muestra los campos de JOptionPane, la tabla 17.9 muestra sus constructores y la tabla 17.10 sus métodos.

Tabla 17.8. Campos de la clase JOptionPane.

Campo	Descripción
static int CANCEL_OPTION	El valor de retorno para el método de clase si se selecciona CANCEL.
static int CLOSED_OPTION	El valor de retorno del método de clases si el usuario cierra la ventana sin seleccionar nada.
static int DEFAULT_OPTION	La apariencia no debería proporcionar ninguna acción, sino utilizar únicamente las opciones del JOptionPane.
static int ERROR_MESSAGE	Utilizado para mensajes de error.
protected Icon icon	El icono utilizado en el panel.
static String ICON_PROPERTY	El nombre de la propiedad ligada por el icono.
static int INFORMATION _MESSAGE	Usado para mensajes de información.
static String INITIAL_ SELECTION_VALUE_PROPERTY	Nombre de la propiedad ligada para initialSelectionValue.
static String INITIAL_VALUE _PROPERTY	Nombre de la propiedad ligada para initialValue.
protected Object initial SelectionValue	El valor inicial para seleccionar en selectionValues.
protected Object initialValue	El valor que debería estar seleccionado inicialmente en las opciones.
static String INPUT_VALUE _PROPERTY	Nombre la propiedad ligada para inputValue.
protected Object inputValue	El valor introducido por el usuario.
protected Object message	El mensaje a mostrar.
static String MESSAGE _PROPERTY	Nombre de la propiedad ligada por el mensaje.

Campo	Descripción
static String MESSAGE_TYPE _PROPERTY	Nombre de la propiedad ligada por el tipo.
protected int messageType	El tipo de mensaje.
static int NO_OPTION	El valor de retorno del método de clase si no se pulsa ningún botón.
static int OK_CANCEL_OPTION	El tipo utilizado para showConfirmDialog.
static int OK_OPTION	El valor de retorno del método de clase si selecciona OK.
static String OPTION_TYPE _PROPERTY	Nombre la propiedad ligada para optionType.
protected Object[] options	Las opciones que mostrar al usuario.
static String OPTIONS _PROPERTY	Nombre de la propiedad ligada para la opción.
protected int optionType	El tipo de acción (DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION o OK_CANCEL_OPTION.).
static int PLAIN_MESSAGE	Indica que no deberían utilizarse iconos.
static int QUESTION_MESSAGE	Utilizado para preguntas.
static String SELECTION _VALUES_PROPERTY	Nombre de la propiedad ligada para selectionValues.
protected Object[] selectionValues	La matriz de valores que el usuario puede seleccionar.
static Object UNINITIALIZED _VALUE	Indica que el usuario aún no ha seleccionado un valor.
protected Object value	El valor seleccionado en curso. Será una opción válida, UNINITIALIZED_VALUE o nulo.
static String VALUE_PROPERTY	Nombre de la propiedad ligada para el valor.
static String WANTS_INPUT _PROPERTY	Nombre de la propiedad ligada para wantsInput.

Campo	Descripción
protected boolean wantsInput	Si true, un componente de la IU se proporcionará al usuario para obtener su entrada.
static int WARNING_MESSAGE	Usado para mensajes de aviso.
static int YES_NO_CANCEL _OPTION	El tipo usado para showConfirmDialog.
static int YES_NO_OPTION	El tipo usado para showConfirmDialog.
static int YES_OPTION	El valor de retorno del método de clase si selecciona YES.

Tabla 17.9. Constructores de la clase JOptionPane.

Constructor	Descripción
JOptionPane()	Construye un JOptionPane con un mensaje de prueba.
JOptionPane(Object message)	Construye una instancia de JOption-Pane para mostrar un mensaje con un tipo de mensaje de texto sencillo y las opciones predeterminadas introducidas por la IU.
JOptionPane(Object message, int messageType)	Construye una instancia de JOptionPa-ne para visualizar el mensaje, el tipo de mensaje indicado y las opciones predeterminadas.
JOptionPane(Object message, int messageType, int optionType)	Construye una instancia de JOptionPa-ne para mostrar un mensaje con el tipo de mensaje y opciones indicadas.
JOptionPane(Object message, int messageType, int optionType, Icon icon)	Construye una instancia de JOptionPa-ne para mostrar el mensaje, el tipo de mensaje indicado, las opciones e icono indicados.
JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options)	Construye una instancia de JOptionPa-ne para mostrar mensaje, el tipo de mensaje, icono y acciones indicadas.
JOptionPane(Object message, int messageType, int option Type, Icon icon, Object[] options, Object initialValue)	Construye una instancia de JOptionPa-ne para mostrar mensaje, el tipo de mensaje, icono y opciones indicadas, con la opción indicada seleccionada.

Tabla 17.10. Métodos de la clase JOptionPane.

Método	Descripción
JDialog createDialog (Component parentComponent, String title)	Construye y devuelve un nuevo JDialog.
JInternalFrame createInternal Frame(Component parent Component, String title)	Construye y devuelve una instancia de JInternalFrame.
AccessibleContext get AccessibleContext()	Obtiene el AccessibleContext.
static JDesktopPane getDesktop PaneForComponent(Component parentComponent)	Obtiene el panel de escritorio del componente indicado.
static Frame getFrameFor Component(Component parentComponent)	Obtiene el marco de componente indicado.
Icon getIcon()	Obtiene el icono que muestra el panel.
Object getInitialSelectionValue()	Obtiene el valor de selección inicial.
Object getInitialValue()	Obtiene el valor inicial.
Object getInputValue()	Obtiene el valor introducido por el usuario, si wantsInput es true.
int getMaxCharactersPerLineCount()	Obtiene el número máximo de caracteres emplazado en la línea de mensaje.
Object getMessage()	Obtiene el objeto mensaje que muestra el panel.
int getMessageType()	Obtiene el tipo de mensaje.
Object[] getOptions()	Obtiene las opciones que puede realizar el usuario.
int getOptionType()	Obtiene el tipo de opciones mostradas.
static Frame getRootFrame()	Obtiene el marco a utilizar para los métodos de clase cuando no se proporcione marco.
Object[] getSelectionValues()	Obtiene los valores de selección.
OptionPaneUI getUI()	Obtiene el objeto IU que implementa la apariencia de este componente.

Método	Descripción
String getUIClassID()	Obtiene el nombre de la clase IU que implementa la apariencia para este componente.
Object getValue()	Obtiene el valor seleccionado por el usuario.
boolean getWantsInput()	Devuelve true si se proporciona al usuario un parentComponent para la entrada.
protected String paramString()	Obtiene una representación de cadena de este JOptionPane.
void selectInitialValue()	Solicita que el valor inicial esté seleccionado, lo que dará el foco al valor inicial.
void setIcon(Icon newIcon)	Asigna el icono a mostrar.
void setInitialSelectionValue (Object newValue)	Asigna el valor de selección inicial.
void setInitialValue(Object newInitialValue)	Asigna el estado habilitado como valor inicial.
void setInputValue(Object newValue)	Asigna el valor de entrada del usuario.
void setMessage(Object newMessage)	Asigna el objeto mensaje del panel de opción.
void setMessageType(int newType)	Asigna el tipo de mensaje del panel de opción.
void setOptions(Object[] new Options)	Asigna las opciones que muestra este panel.
void setOptionType(int newType)	Asigna las opciones a mostrar.
static void setRootFrame (Frame newRootFrame)	Asigna el marco a utilizar para los métodos de clase en que un marco no se proporciona.
void setSelectionValues(Object[] newValues)	Asigna los valores de selección para un panel que proporciona al usuario una lista de elementos para seleccionar.
void setUI(OptionPaneUI ui)	Asigna el objeto que implementa la apariencia de este componente.

Método	Descripción
void setValue(Object newValue)	Asigna el valor seleccionado por el usuario.
void setWantsInput(boolean newValue)	Si newValue es true, se añade un componente para permitir al usuario introducir un valor.
static int showConfirmDialog (Component parentComponent, Object message)	Muestra un cuadro de diálogo modal con las opciones Yes, No y Cancel.
static int showConfirmDialog (Component parentComponent, Object message, String title, int optionType)	Muestra un cuadro de diálogo modal donde el número de opciones está determinado por el parámetro optionType.
static int showConfirmDialog V	Muestra dónde el parámetro messageType determina el icono para mostrar.
static int showConfirmDialog (Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon)	Muestra un cuadro de diálogo modal, con el icono indicado, donde el número de opciones está determinado por el parámetro optionType.
static String showInputDialog (Component parentComponent, Object message)	Muestra un cuadro diálogo de mensaje de pregunta solicitando entrada por el usuario; tiene como padre parentComponent.
static String showInputDialog (Component parentComponent, Object message, String title, int messageType)	Muestra un cuadro de diálogo solicitando entrada al usuario, tiene como padre parentComponent con el cuadro de diálogo con un título title y messageType.
static Object showInputDialog (Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue)	Solicita entrada de usuario en un cuadro de diálogo bloqueado donde la selección inicial, selecciones posibles y demás opciones son las indicadas.
static String showInputDialog (Object message)	Muestra un cuadro de diálogo de mensaje de pregunta solicitando entrada al usuario.

Método	Descripción
static int showInternalConfirm Dialog(Component parent Component, Object message)	Muestra un cuadro de diálogo interno con las opciones Yes, No y Cancel.
static int showInternalConfirm Dialog(Component parent Component, Object message, String title, int optionType)	Muestra un cuadro de diálogo interno con el número de opciones predeterminado por el parámetro optionType.
static int showInternalConfirm Dialog(Component parent Component, Object message, String title, int optionType, int messageType)	Muestra un cuadro de diálogo interno de número de opciones predeterminado por el parámetro optionType, donde el mensaje y el parámetro messageType determinan el icono a mostrar.
static int showInternalConfirm Dialog(Component parent Component, Object message, String title, int optionType, int messageType, Icon icon)	Muestra un cuadro de diálogo interno con un icono indicado, donde el número de opciones es determinado por el parámetro optionType.
static String showInternalInput Dialog(Component parent Component, Object message)	Muestra un cuadro de diálogo de mensaje interno de pregunta solicitando entrada del usuario que tiene como padre parentComponent.
static String showInternalInput Dialog(Component parent Component, Object message, String title, int messageType)	Muestra el cuadro de diálogo de entrada interno para el usuario que tiene como padre parentComponent, con el cuadro de diálogo con título title y el tipo de mensaje messageType.
static Object showInternalInput Dialog(Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue)	Solicita entrada de usuario en un cuadro de diálogo interno bloqueado donde la selección inicial, las selecciones posibles y demás opciones son las indicadas.
static void showInternalMessage Dialog(Component parentComponent, Object message)	Muestra un cuadro de diálogo de confirmación interno.
static void showInternalMessage Dialog(Component parentComponent, Object message, String title, int messageType)	Muestra un cuadro de diálogo interno que muestra un mensaje utilizando un icono asignado por el parámetro messageType.

Método	Descripción
static void showInternalMessage Dialog(Component parentCompo-nent, Object message, String title, int messageType, Icon icon)	Muestra un cuadro de diálogo interno que visualiza un mensaje y especifica todos los parámetros.
static int showInternalOptionDialog (Component parentComponent, Ob-ject message, String title, int op-tionType, int messageType, Icon icon, Object[] options, Object ini-tialValue)	Muestra un cuadro de diálogo interno con el icono indicado, donde las opcio-nes iniciales están asignadas por el pa-rámetro initialValue y el número de op-ciones predeterminado por el parámetro optionType.
static void showMessageDialog (Component parentComponent, Object message)	Muestra un cuadro de diálogo de con-firmación.
static void showMessageDialog (Component parentComponent, Object message, String title, int messageType)	Muestra un cuadro de diálogo que visua-liza un mensaje utilizando un icono pre-determinado asignado por el parámetro messageType.
static void showMessageDialog (Component parentComponent, Ob-ject message, String title, int me-ssageType, Icon icon)	Muestra un cuadro de diálogo que vi-sualiza un mensaje y especifica todos los parámetros.
static int showOptionDialog(Com-ponent parentComponent, Object message, String title, int optionTy-pe, int messageType, Icon icon, Object[] options, Object initialValue)	Muestra un cuadro de diálogo modal con un icono indicado, donde la opción inicial queda determinada por el pará-metro initialValue y el número de opcio-nes queda asignado por el parámetro optionType.
void updateUI()	Llamada por UIManager cuando cambia la apariencia.

Puede utilizar los métodos estáticos de JOptionPane (es decir, métodos de clase que invocamos directamente utilizando la clase, como sigue: JOptionPane.showMessageDialog()) para visualizar cuatro tipos de cuadro de diálogo. Aquí tiene sus métodos y los cuadros de diálogo que crean:

- showMessageDialog(). Muestra algún mensaje al usuario.
- showConfirmDialog. Solicita la confirmación a una pregunta y recibe una respuesta sí/no/cancelar.

- showInputDialog. Solicita entrada de usuario.
- showOptionDialog. Un cuadro de diálogo configurable.

Para cada uno de estos métodos, existe también una versión de marco interna (de poco peso), como showInternalMessageDialog, showInternalInputDialog y así sucesivamente, en el caso de que quiera ajustarse a marcos internos para conservar los recursos del sistema.

Todos estos métodos pueden devolver un entero que indica el botón que seleccionó el usuario; los valores posibles son YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION y CLOSED_OPTION. Estos cuadros de diálogo son *modales*, lo que indica que el usuario debe cerrarlos antes de continuar con el resto el programa (si hace clic sobre cualquier otra ventana del programa únicamente genera un pitido).

Todo esto se comprende mejor con ejemplos, por lo que analizaremos esta clase utilizando varios ejemplos a lo largo de los siguientes temas.

Crear cuadros de diálogo con panel de opciones de confirmación

El especialista de soporte dice, "Necesitamos asegurar que los usuarios están de acuerdo con esto". Preguntamos, "¿Necesitamos asegurar que ellos están conformes con visualizar la suma después de que hayan pulsado el botón Añadir?" "Utilice un cuadro de diálogo de confirmación", dice el ESP.

Los cuadros de diálogo de confirmación permiten al usuario hacer clic sobre botones para confirmar o detener una acción. Utilizamos el método estático showConfirmDialog de JOptionPane para crear esta clase de cuadro de diálogo. Aquí tiene las constantes de JOptionPane que puede utilizar con este método, indicando los botones que quiere mostrar: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION y OK_CANCEL_OPTION.

Veamos un ejemplo en el que visualizaremos el cuadro de diálogo de confirmación con un botón Sí y otro No. Para visualizar el cuadro de diálogo, sitúe un botón en un applet, y proporcione al botón el título **Mostrar diálogo**. Cuando se pulsa el botón, utilice showConfirmDialog() para mostrar el nuevo cuadro de diálogo y utilice el valor de retorno del método para indicar el botón que fue pulsado mostrando un mensaje en la barra de estado del applet. Aquí tiene el código:

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

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

    /*
<APPLET
    CODE = dialogconfirm.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
    */

public class dialogconfirm extends JApplet
{
    JWindow jwindow = new JWindow();

    public void init()
    {
        final Container contentPane = getContentPane();
        JButton jbutton = new JButton("Mostrar diálogo");

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jbutton);

        jbutton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
               int result = JOptionPane.showConfirmDialog((Component)
                   null, "Seleccione Sí o No", "Seleccione Sí o No",
                   JOptionPane.YES_NO_OPTION);

               if (result == JOptionPane.YES_OPTION) {
                   showStatus("Hizo clic sobre Sí.");
               } else {
                   showStatus("Hizo clic sobre No.");
               }
            }
        });
    }
}
```

La figura 17.5 muestra el resultado de este código, donde puede ver el cuadro de diálogo sí/no. Si se pulsa un botón se cerrará el cuadro de diálogo y el applet indicará el botón que se pulsó con un mensaje en su barra de estado. Este programa se encuentra en el archivo dialogconfirm.java del CD.

Crear cuadros de diálogo con panel de opciones de mensaje

"No, no", dice el especialista de soporte de producto. "No quise decir que debería utilizar un cuadro de diálogo de confirmación ahí; debería ser un cuadro de diálogo de mensaje". "Bien", decimos, "espero que esté seguro".

Los cuadros de diálogo de mensaje nos permiten mostrar un mensaje al usuario y utilizar el método estático showMessageDialog() de la clase JOptionPane para crear esta clase de cuadros de diálogo, pasando el mensaje que queremos mostrar a ese método.

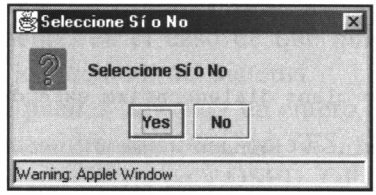


Figura 17.5. Un cuadro de diálogo de confirmación.

Puede seleccionar el tipo de icono que quiere visualizar en estos cuadros de diálogo; aquí tiene las constantes de JOptionPane que puede utilizar con este método para indicar el icono que quiere mostrar: INFORMATION_MESSAGE, ERROR_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE y PLAIN_MESSAGE.

Aquí tiene un ejemplo que le muestra cómo utilizar todos estos tipos de cuadro de diálogo de mensaje. En este caso, añada un botón para cada tipo de cuadro de diálogo y haga clic sobre el botón para visualizar el cuadro de diálogo correspondiente:

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

    /*
<APPLET
    CODE = dialogmessage.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class dialogmessage extends JApplet implements ActionListener
{
    JButton jbutton1 = new JButton("Mostrar diálogo de información");
    JButton jbutton2 = new JButton("Mostrar diálogo de error");
    JButton jbutton3 = new JButton("Mostrar diálogo de aviso");
    JButton jbutton4 = new JButton("Mostrar diálogo de pregunta");
    JButton jbutton5 = new JButton("Mostrar diálogo simple");

    public void init()
    {
        Container contentPane = getContentPane();

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

```
        contentPane.add(jbutton1);
        contentPane.add(jbutton2);
        contentPane.add(jbutton3);
        contentPane.add(jbutton4);
        contentPane.add(jbutton5);

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this);
        jbutton4.addActionListener(this);
        jbutton5.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
            String dialogtitle = "Diálogo";
            String dialogmessage = "¡Hola desde Swing!";
            int dialogtype = JOptionPane.PLAIN_MESSAGE;

            if(e.getSource() == jbutton1) {
                dialogtype = JOptionPane.INFORMATION_MESSAGE;
            } else if(e.getSource() == jbutton2) {
                dialogtype = JOptionPane.ERROR_MESSAGE;
            } else if(e.getSource() == jbutton3) {
                dialogtype = JOptionPane.WARNING_MESSAGE;
            } else if(e.getSource() == jbutton4) {
                dialogtype = JOptionPane.QUESTION_MESSAGE;
            } else if(e.getSource() == jbutton5) {
                dialogtype = JOptionPane.PLAIN_MESSAGE;
            }

            JOptionPane.showMessageDialog((Component) null,
                  dialogmessage, dialogtitle, dialogtype);
    }
}
```

La figura 17.6 muestra el resultado de este código, donde puede ver los botones en el applet que representan los tipos de cuadro de diálogo de mensaje posibles, así como un cuadro de diálogo de información. Este ejemplo se encuentra el archivo dialogmessage.java del CD.

Crear cuadros de diálogo con panel de opciones de campo de texto de entrada

"No", dice el especialista de soporte a productos, "eso no tiene nada que ver con lo que yo quería. Yo no quería un cuadro de diálogo de mensaje; quería un cuadro de diálogo de entrada para permitir al usuario introducir algún texto". "Ya", decimos.

Puede utilizar el método showInputDialog de la clase JOptionPane para mostrar un cuadro de diálogo con un campo de texto que permite al usuario introducir texto. Este método devuelve el texto que el usuario introdujo en el campo de texto o *null* si hizo clic sobre el botón Cancelar.

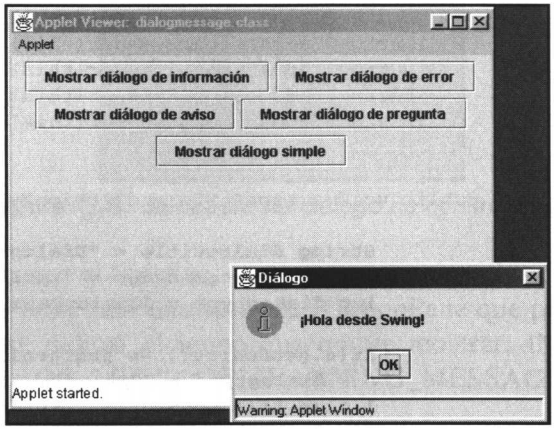


Figura 17.6. Creación de distintos tipos de cuadros de diálogo de mensaje.

Aquí tiene un ejemplo en el que leemos algún texto de entrada del usuario y mostramos ese texto en la barra de estado del applet. El código es bastante sencillo. Aquí lo tiene:

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

    /*
<APPLET
    CODE = dialogtext.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
    */

public class dialogtext extends JApplet implements ActionListener
{
    JButton jbutton = new JButton("Mostrar diálogo");
    String message = "Introduzca el texto";

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jbutton);
```

```
        jbutton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        String result = JOptionPane.showInputDialog(message);

        if(result == null)
            showStatus("Hizo clic sobre Cancelar");
        else
            showStatus("Ha escrito: " + result);
    }
}
```

La figura 17.7 muestra resultado el de este código y puede ver el cuadro de diálogo de entrada. Cuando el usuario hace clic sobre el botón **Aceptar**, el cuadro de diálogo desaparece y el texto del campo de entrada del cuadro de diálogo aparece en la barra de estado del applet. Eso es todo. Este ejemplo se encuentra en el archivo dialogtext.java del CD.

Observe que puede utilizar ahowInputDialog también para otros usos; por ejemplo, puede mostrar un cuadro combinado en un cuadro de diálogo. Consulte el siguiente tema para más detalles.

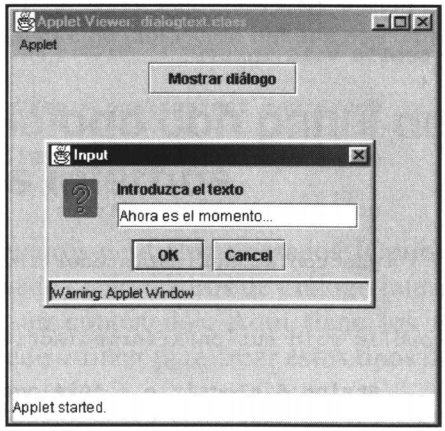


Figura 17.7. Un cuadro de diálogo básico de entrada de texto.

Crear cuadros de diálogo con panel de opciones para entrada de cuadros combinados

"No, no", dice el especialista de soporte a productos, "no quería un campo de texto en ese cuadro de diálogo, quería un..." "¿Cuadro combinado?", preguntamos. "¿Cómo lo supo?", pregunta el ESP. "Telepatía", decimos.

Puede utilizar el método showInputDialog de la clase JOptionPane para mostrar una gran variedad de controles; en este tema, utilizamos un cuadro combinado. Para hacerlo, pase un cuadro combinado a showInputDialog así como una matriz para contener los elementos que desea en el cuadro combinado y el elemento que quiere seleccionado por defecto. Este caso muestra varios postres y permite al usuario seleccionar uno. Cuando el usuario hace clic sobre el botón **Aceptar**, se muestra la selección en la barra de estado del applet. Aquí tiene el código:

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

    /*
<APPLET
    CODE = dialogcombo.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
    */

public class dialogcombo extends JApplet implements ActionListener
{
    JButton jbutton = new JButton("Mostrar diálogo");

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());

        contentPane.add(jbutton);

        jbutton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        String dialogtitle = "Diálogo";
        String dialogmessage = "¿Cuál le gusta más?";
        String[] desserts = {
            "tarta de queso", "helado", "mousse", "pastel"
        };

        String dessert = (String)JOptionPane.showInputDialog(
            dialogcombo.this, dialogmessage, dialogtitle,
            JOptionPane.QUESTION_MESSAGE, null,
            desserts, desserts[0]);

        if ( dessert == null )
            showStatus ("Pulsó Cancelar");
        else
```

```
                showStatus ("Su postre favorito: " + dessert );
        }
    }
```

La figura 17.8 muestra el resultado de este código, donde puede ver el cuadro combinado en el cuadro de diálogo. Cuando el usuario selecciona un elemento del cuadro combinado y hace clic sobre Aceptar, el cuadro de diálogo se cierra y el elemento seleccionado aparece en la barra de estado del applet.

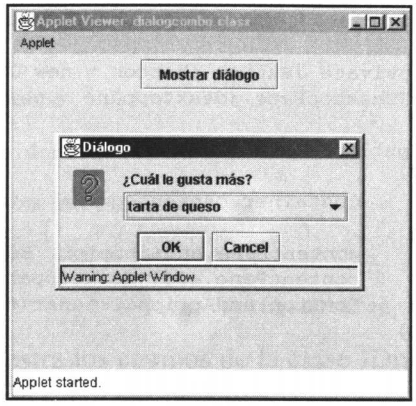


Figura 17.8. Un cuadro de diálogo de entrada con un cuadro combinado.

Crear cuadros de diálogo con panel de opciones de marcos internos

Además de los cuadros de diálogo pesados JOptionPane creados en los ejemplos previos, puede crear cuadros de diálogo internos de poco peso que aparecen dentro de un contenedor. Aquí tiene los métodos estáticos de JOptionPane que puede utilizar para crear estos tipos de cuadro de diálogo:

- ***showInternalMessageDialog***. Muestra algún mensaje al usuario.
- ***showInternalConfirmDialog***. Solicita la confirmación a una pregunta y recibe una respuesta sí/no/cancelar.
- ***showInternalInputDialog***. Solicita entrada al usuario.
- ***showInternalOptionDialog***. Un cuadro de diálogo configurable.

Aquí tiene un ejemplo en el que se visualiza un cuadro de diálogo de mensaje interno en un panel de escritorio cuando el usuario hace clic sobre el botón **Mostrar diálogo interno**. Éste es el código:

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

    /*
<APPLET
    CODE = optionpaneinternal.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class optionpaneinternal extends JApplet implements ActionListener
{
    private JButton jbutton = new JButton("Mostrar diálogo interno");
    JDesktopPane jdesktoppane = new JDesktopPane();

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.add(jbutton, BorderLayout.SOUTH);
        contentPane.add(jdesktoppane);
        jbutton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showInternalMessageDialog(
            jdesktoppane, "¡Hola desde Swing!", "Diálogo",
            JOptionPane.INFORMATION_MESSAGE);
    }
}
```

La figura 17.9 muestra el resultado, donde puede ver el cuadro de diálogo de mensaje interno. Este ejemplo se encuentra en el archivo optionpaneinternal.java del CD.

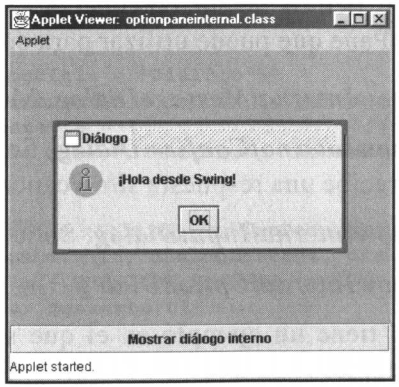


Figura 17.9. Un diálogo de mensaje interno.

Crear cuadros de diálogo con JDialog

"Bien", dice el programador novato, "los cuadros de diálogo sencillos JOptionPane son muy bonitos, pero quiero crear mis propios cuadros de diálogo". "¿Desde cero?", preguntamos. "Desde cero", dice el PN. "Entonces quiere la clase JDialog", decimos.

La clase JDialog es la base de los cuadros de diálogo personalizados en la Swing. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-java.awt.Window
                    |
                    +-java.awt.Dialog
                          |
                          +-javax.swing.JDialog
```

La tabla 17.11 muestra los campos de la clase JDialog, la tabla 17.12 sus constructores y la tabla 17.13 sus métodos.

Tabla 17.11. Campos de la clase JDialog.

Campo	Descripción
protected AccessibleContext accessibleContext	El contexto accesible.
protected JRootPane rootPane	El panel raíz.
protected boolean rootPaneCheckingEnabled	Determina si el panel raíz se puede comprobar.

Tabla 17.12. Constructores de la clase JDialog.

Constructor	Descripción
JDialog()	Construye un cuadro de diálogo no modal.
JDialog(Dialog owner)	Construye un cuadro de diálogo no modal con el cuadro de diálogo dado como padre.
JDialog(Dialog owner, boolean modal)	Construye un cuadro de diálogo modal o no modal, sin título y con el

Constructor	Descripción
	cuadro de diálogo indicado como padre.
JDialog(Dialog owner, String title)	Construye un cuadro de diálogo no modal con el título indicado y con el marco indicado como padre.
JDialog(Dialog owner, String title, boolean modal)	Construye un cuadro de diálogo modal o no modal con el título indicado y con el marco padre indicado.
JDialog(Frame owner)	Construye un cuadro de diálogo no modal sin título con el marco especificado como padre.
JDialog(Frame owner, boolean modal)	Construye un cuadro de diálogo modal o no modal sin título y con el marco indicado como padre.
JDialog(Frame owner, String title)	Construye un cuadro de diálogo no modal con el título indicado y con el marco indicado como padre.
JDialog(Frame owner, String title, boolean modal)	Construye un cuadro de diálogo modal o no modal con el título indicado y el marco indicado como padre.

Tabla 17.13. Métodos de la clase JDialog.

Método	Descripción
protected void addImpl(Component comp, Object constraints, int index)	Añade hijos a contentPane.
protected JRootPane createRootPane()	Llamado por los métodos constructores para crear el JRootPane predeterminado.
protected void dialogInit()	Llamado por los constructores para inicializar la propiedad JDialog.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
Container getContentPane()	Obtiene el contentPane.
int getDefaultCloseOperation()	Obtiene la operación que sucede cuando el usuario inicia una operación de cierre sobre este cuadro de diálogo.

Método	Descripción
Component getGlassPane()	Obtiene el objeto glassPane para este cuadro de diálogo.
JMenuBar getJMenuBar()	Obtiene la barra de menú asignada a este cuadro de diálogo.
JLayeredPane getLayeredPane()	Obtiene el objeto layeredPane para este cuadro de diálogo.
JRootPane getRootPane()	Obtiene el objeto rootPane para este cuadro de diálogo.
protected boolean isRootPaneCheckingEnabled()	Devuelve true si el panel raíz se puede comprobar.
protected String paramString()	Obtiene una representación de cadena de este JDialog.
protected void processKeyEvent (KeyEvent e)	Procesa los eventos de teclado que suceden sobre este JDialog.
protected void processWindowEvent (WindowEvent e)	Procesa los eventos de ventana dependiendo del estado de la propiedad defaultCloseOperation.
void remove(Component comp)	Elimina el componente indicado de este contenedor.
void setContentPane(Container contentPane)	Asigna la propiedad contentPane.
void setDefaultCloseOperation(int operation)	Asigna la operación que sucederá por defecto si el usuario cierra el cuadro de diálogo.
void setGlassPane(Component glassPane)	Asigna la propiedad glassPane.
void setJMenuBar(JMenuBar menu)	Asigna la barra de menú para este cuadro de diálogo.
JLayeredPane getLayeredPane()	Asigna la propiedad layeredPane.
void setLayout(LayoutManager manager)	Asigna la distribución del contentPane.
void setLocationRelativeTo(Component c)	Asigna la localización del cuadro de diálogo relativa al componente indicado.

Método	Descripción
protected void setRootPane(JRoot-Pane root)	Asigna la propiedad rootPane.
protected void setRootPaneCheck-ingEnabled(boolean enabled)	Devuelve true si las llamadas a add() y setLayout() pueden generar una excepción.
void update(Graphics g)	Simplemente llama a paint(g).

Crearemos un ejemplo de cuadro de diálogo utilizando la clase JDialog. En este caso, haremos el cuadro de diálogo modal pasando un parámetro final con el valor *true* al constructor JDialog y añadiremos botones al cuadro de diálogo: **Aceptar** y **Cancelar**. Al añadir receptores de acciones a los botones, podemos saber qué botón se pulsó y podemos indicarlo en la barra de estado del applet. Una vez se ha pulsado un botón, llamamos al método *dispose* del cuadro de diálogo para eliminarlo. Aquí tiene el código (observe que al igual que otras ventanas de la Swing, puede añadir controles al objeto JDialog utilizando su panel de contenidos):

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

    /*
<APPLET
    CODE = dialog.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
    */

public class dialog extends JApplet implements ActionListener
{
     JLabel jlabel = new JLabel("Haga clic sobre el botón Aceptar o
Cancelar");

    JButton jbutton1 = new JButton("Mostrar diálogo"),
        jbutton2 = new JButton("Aceptar"),
        jbutton3 = new JButton("Cancelar");

    private JDialog dialog = new JDialog((Frame) null,
        "Diálogo", true);

    public void init()
    {
        Container contentPane = getContentPane();
        Container dialogContentPane = dialog.getContentPane();

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

```
        contentPane.add(jbutton1);

        dialogContentPane.setLayout(new FlowLayout());

        dialogContentPane.add(jlabel);
        dialogContentPane.add(jbutton2);
        dialogContentPane.add(jbutton3);

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            dialog.setBounds(200, 200, 200, 120);
            dialog.show();
        } else if(e.getSource() == jbutton2) {
            showStatus("Hizo clic sobre Aceptar");
            dialog.dispose();
        } else if(e.getSource() == jbutton3) {
            showStatus("Hizo clic sobre Cancelar");
            dialog.dispose();
        }
    }
}
```

La figura 17.10 muestra el resultado de este código. Como puede ver, el cuadro de diálogo visualiza los botones **Aceptar** y **Cancelar**. Si hace clic sobre un botón cierra el cuadro de diálogo y aparece un mensaje en la barra de estado del applet indicando el botón pulsado. Eso es todo. Este ejemplo se encuentra en el archivo dialog.java del CD.

Observe que el cuadro de diálogo que muestra la figura 17.10 únicamente acepta entradas de botón del usuario; para observar un ejemplo más avanzado, consulte el tema siguiente.

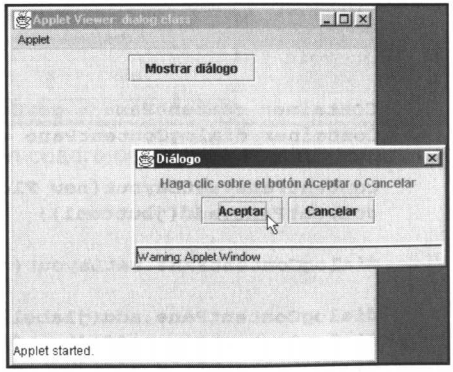


Figura 17.10. Un cuadro de diálogo JDialog básico.

Obtener entrada de los cuadros de diálogo creados con JDialog

Puede configurar los cuadros de diálogo creados con la clase JDialog como desee; examinaremos un ejemplo en este tema. Aqu, mostraremos un campo de texto en el que el usuario puede introducir texto. Si el cuadro de diálogo se cierra usando el botón **Aceptar**, el código mostrará el texto que introdujo el usuario en el campo de texto de la barra de estado del applet; en cualquier otro caso, si el usuario hizo clic sobre el botón **Cancelar**, el código muestra el mensaje "Seleccionó Cancelar". Aquí tiene el código:

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

    /*
<APPLET
    CODE = dialoginput.class
    WIDTH = 350
    HEIGHT = 280>
<A/PPLET>
    */

public class dialoginput extends JApplet implements ActionListener
{
    JLabel jlabel = new JLabel("Introduzca el texto:");

    JTextField jtextfield = new JTextField(15);

    JButton jbutton1 = new JButton("Mostrar diálogo"),
        jbutton2 = new JButton("Aceptar"),
        jbutton3 = new JButton("Cancelar");

    private JDialog dialog = new JDialog((Frame) null,
        "Diálogo", true);

    public void init()
    {
        Container contentPane = getContentPane();
        Container dialogContentPane = dialog.getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(jbutton1);

        dialogContentPane.setLayout(new FlowLayout());

        dialogContentPane.add(jlabel);
        dialogContentPane.add(jtextfield);
        dialogContentPane.add(jbutton2);
```

```
        dialogContentPane.add(jbutton3);

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            dialog.setBounds(200, 200, 200, 150);
            dialog.show();
        } else if(e.getSource() == jbutton2) {
            showStatus(jtextfield.getText());
            dialog.dispose();
        } else if(e.getSource() == jbutton3) {
                showStatus("Hizo clic sobre Cancelar");
                dialog.dispose();
        }
    }
}
```

La figura 17.11 muestra el resultado de este código, donde puede ver el cuadro de diálogo con el campo de texto. Cuando el usuario introduce texto y hace clic sobre **Aceptar**, el texto aparece en la barra de estado del applet. Es todo. Este ejemplo se encuentra en el archivo dialoginput.java del CD.

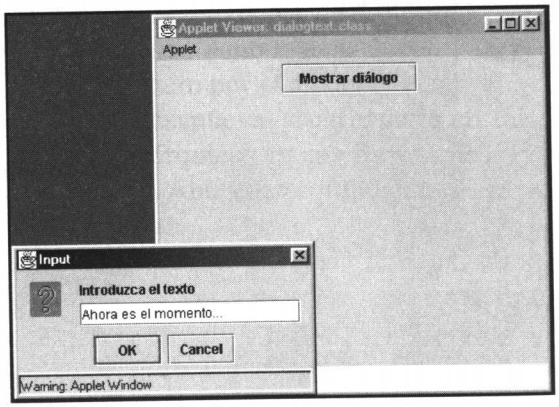


Figura 17.11. Un cuadro de diálogo JDialog con un campo de texto.

18 Swing: tablas y árboles

En este capítulo, examinaremos dos de los componentes más importantes de la Swing: tablas y árboles. Ambos temas tienen un ámbito muy grande y requerirían un volumen entero por sí mismos. Puede adivinar su función por sus nombres. Las tablas presentan al usuario datos en una forma tabular y los árboles muestran datos jerárquicos en una forma que ha pasado a ser común: usando nodos expandibles. Antes de profundizar en estos temas, veremos un breve resumen de cada uno.

Tablas

Las tablas pueden ser el componente más complejo en la Swing. De hecho, se dedica un paquete completo a ellas: swing.table. Las tablas presentan y permiten al usuario editar datos distribuidos en filas y columnas. En la Swing, las tablas están compuestas por cabeceras de columna y objetos columna. Por supuesto, también puede manejar los datos por filas y por celdas individuales, aunque estas construcciones no son objetos.

Las tablas de la Swing son muy flexibles, y por esta misma razón son muy complejas. Puede editar los datos en la tabla directamente y utilizar varios modos de selección (incluyendo columna, fila y selección de celda indivi-

dual). Puede utilizar su propio modelo de columnas y cabeceras de columna, tablas decoradas, dibujar sus propias celdas derivando un nuevo renderizador de celda a partir de la interfaz TableCellRenderer, manejar la edición de celdas implementando la interfaz CellEditor, usar acciones de pulsaciones de teclas, reordenar columnas, cambiar el tamaño de columnas y mucho más.

Como con otros componentes de la Swing, puede personalizar casi todo en una tabla y debido a que involucra muchos subcomponentes, la parte el proceso de personalización puede llegar a ser muy compleja.

Árboles

Uno de los dos componentes en la programación IU que permite al usuario manejar datos es el *árbol*. Si ha utilizado el explorador de *Microsoft Windows*, ya conocerá los árboles; esta utilidad presenta una estructura de directorio usando un árbol. Los árboles utilizan la analogía de "carpetas y hojas" para presentar los datos; cuando hace doble clic sobre una carpeta visible, aparecen las páginas (u *hojas*), así como cualquiera de sus carpetas dependientes. También puede cerrar las carpetas de nuevo haciendo doble clic sobre ellas. Los árboles distribuyen sus datos en una jerarquía vertical, conectada con líneas, facilitando su presentación desplazable. A veces también se utilizan para visualizar directorios (como carpetas) y archivos (como hojas) de una forma manejable.

Como las tablas, los árboles son suficientemente complejos como para tener su propio paquete en la Swing: swing.tree. En términos de programación, los árboles están compuestos de *nodos* y cada nodo puede ser una carpeta o una hoja. Las carpetas pueden tener nodos hijos. Todos los nodos excepto el nodo raíz tienen un único nodo padre. Como puede suponer, la presentación de las carpetas y hojas depende de la apariencia. Cuando abre una carpeta, se considera *expandida* y cuando la cierra, se *colapsa*. Como con las tablas, puede personalizar los árboles con sus propios renderizadores y editores. En este capítulo obtendrá una buena introducción al tema.

Esto es suficiente como resumen de lo que aparece en este capítulo. Vendrá mucho más: las tablas y los árboles representan una cantidad enorme de profundidad de programación.

Crear tablas

El gran jefe aparece y dice, "Sobre el nuevo programa de hoja de cálculo que está escribiendo..." "No sabía que estuviera escribiendo nada de eso",

decimos. "No interrumpa", dice el gran jefe. "Lo necesitábamos ayer. ¿Está hecho?" "Mmh", decimos, "parece un trabajo para la clase JTable".

JTable es el componente de la Swing que presenta los datos en un formato de tabla bidimensional. Aquí tiene el diagrama de herencia para JTable:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JTable
```

La tabla 18.1 muestra los campos de la clase JTable, la tabla 18.2 sus constructores y la tabla 18.3 sus métodos (observe lo extensa que es ésta clase).

Tabla 18.1. Campos de la clase JTable.

Campo	Descripción
static int AUTO_RESIZE_ALL_COLUMNS	Cambio de tamaño de la columna durante las operaciones de cambio de tamaño.
static int AUTO_RESIZE_LAST_COLUMN	También la última columna únicamente durante todas las operaciones de cambio de tamaño.
static int AUTO_RESIZE_NEXT_COLUMN	Cuando una columna se ajusta, este campo ajusta la siguiente, de forma opuesta.
static int AUTO_RESIZE_OFF	Utiliza una barra de desplazamiento para ajustar el ancho de columna.
static int AUTO_RESIZE_SUBSEQUENT_COLUMNS	Cambia las columnas siguientes para preservar el ancho total hasta el cambio de tamaño.
protected boolean autoCreateColumns-FromModel Consulta el TableModel	Para crear el conjunto de columnas predeterminado si este campo es true.
protected int autoResizeMode	Asigna si la tabla cambia automáticamente el tamaño de la

Campo	Descripción
	anchura de sus columnas para ocupar el ancho total de la tabla.
protected TableCellEditor cellEditor	Un objeto que sobrescribe la celda actual y permite al usuario cambiar sus contenidos.
protected boolean cellSelectionEnabled	Si es true, tanto las filas como las columnas seleccionadas pueden configurarse al mismo tiempo.
protected TableColumnModel columnModel	El TableColumnModel de la tabla.
protected TableModel dataModel	El TableModel de la tabla.
protected Hashtable defaultEditorsByColumnClass	La tabla de objetos que visualiza y edita el contenido de una celda, con índice por clase.
protected Hashtable defaultRenderersByColumnClass	La tabla de objetos que visualiza el contenido de una celda, con índice por clase.
protected int editingColumn	La columna de la celda en edición.
protected int editingRow	La fila de celda editada.
protected Component editorComp	El componente que procesa la edición.
protected Color gridColor	El color de rejilla.
protected Dimension preferredViewportSize	Utilizado por la interfaz Scrollable para determinar el área inicial visible.
protected int rowHeight	La altura de las filas en la tabla.
protected int rowMargin	La altura de las imágenes de filas.
protected boolean rowSelectionAllowed	Devuelve true si se permite selección de fila en esta tabla.

Campo	Descripción
protected Color selectionBackground	El color de fondo de las celdas seleccionadas.
protected Color selectionForeground	El color de primer plano de las celdas seleccionadas.
protected ListSelectionModel selectionModel	El ListSelectionModel de la tabla; se utiliza para controlar las filas seleccionadas.
protected boolean showHorizontalLines	Las líneas horizontales se dibujan entre las celdas cuando este campo es true.
protected boolean showVerticalLines	Las líneas verticales se dibujan entre las celdas si este campo es true.
protected JTableHeader tableHeader	El JTableHeader que funciona con la tabla.

Tabla 18.2. Constructores de la clase JTable.

Constructor	Descripción
JTable()	Construye un JTable predeterminado.
JTable(int numRows, int numColumns)	Construye un JTable con numRows y numColumns de celdas vacías.
JTable(Object[][] rowData, Object[] columnNames)	Construye un JTable para visualizar los valores en una matriz bidimensional, rowData, con nombres de columna columnNames.
JTable(TableModel dm)	Construye un JTable inicializado con dm como modelo de datos, un modelo de columna predeterminado y un modelo de selección predeterminado.
JTable(TableModel dm, TableColumnModel cm)	Construye un JTable inicializado con dm como modelo de datos, cm como modelo de columna y un modelo de selección predeterminado.

Constructor	Descripción
JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)	Construye un JTable inicializado con dm como modelo de datos, cm como modelo de columnas y m como modelo de selección.
JTable(Vector rowData, Vector columnNames)	Construye un JTable para visualizar los valores en el vector de Vectores, rowData, con nombres de columna dados en columnNames.

Tabla 18.3. Métodos de la clase JTable.

Método	Descripción
void addColumn(TableColumn aColumn)	Añade la columna al final de la matriz de columnas.
void addColumnSelectionInterval(int index0, int index1)	Añade las columnas desde index0 a index1, incluida, a la selección.
void addNotify()	Llama a configureEnclosingScrollPane.
void addRowSelectionInterval(int index0, int index1)	Añade las filas desde index0 a index1, incluida, a la selección.
void clearSelection()	Deselecciona todas las columnas y filas seleccionadas.
void columnAdded(TableColumnModelEvent e)	Indica a los receptores que una columna ha sido añadida al modelo.
int columnAtPoint(Point point)	Obtiene el índice de la columna en que point reside (ó -1 si reside fuera de los límites del receptor).
void columnMarginChanged(ChangeEvente)	Notifica a los receptores que una columna se ha movido debido al cambio de imágenes.

Método	Descripción
void columnMoved(TableColumnModel-Event e)	Notifica a los receptores que una columna cambia de posición.
void columnRemoved(TableColumnModel-Event e)	Notifica a los receptores que una columna fue eliminada del modelo.
void columnSelectionChanged(ListSelection-Event e)	Notifica a los receptores que el modelo de selección de TableColumnModel ha cambiado.
protected void configureEnclosingScroll-Pane()	Configura los JScrollPane contenidos instalando el tableHeader de la tabla, columnHeaderView del panel de desplazamiento y así sucesivamente.
int convertColumnIndexToModel(int view-ColumnIndex)	Obtiene el índice de la columna en el modelo cuyos datos se muestran en la columna viewColumnIndex de la pantalla.
int convertColumnIndexToView(int model-ColumnIndex)	Obtiene el índice de la columna en la vista que muestra los datos desde la columna modelColumnIndex del modelo.
protected TableColumnModel createDefaultColumnModel()	Obtiene el modelo de objeto de columna predeterminado, que es un DefaultTableColumnModel.
void createDefaultColumnsFromModel()	Crea columnas predeterminadas para la tabla partir del modelo de datos utilizando los métodos getColumnCount() y getColumnClass() definidos en la interfaz TableModel.
protected TableModel createDefaultData-Model()	Obtiene el modelo de objetos de la tabla por defecto, que es un DefaultTableModel.

Método	Descripción
protected void createDefaultEditors()	Crea editores de celdas predeterminados para objetos, números y valores boolean.
protected void createDefaultRenderers()	Crea renderizadores predeterrminados.
protected ListSelectionModel createDefaultSelectionModel()	Obtiene el modelo de selección de objetos predeterminado, que es un DefaultListSelectionModel.
protected JTableHeader createDefaultTableHeader()	Obtiene el objeto cabecera de tabla predeterminado, que es un JTableHeader.
static JScrollPane createScrollPaneForTable(JTable aTable)	Obsoleto. Reemplazado por el nuevo new JScrollPane (aTable).
boolean editCellAt(int row, int column)	Inicia la adición en la fila situada en row y column, si la celda se puede editar.
boolean editCellAt(int row, int column, Event Object e)	Inicia la edición de la celda row y column, si la celda se puede editar.
void editingCanceled(ChangeEvent e)	Llamado cuando se cancela la edición.
void editingStopped(ChangeEvent e)	Llamado cuando termina la edición.
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext asociado con este JComponent.
boolean getAutoCreateColumnsFromModel()	Obtiene si la tabla creará columnas predeterminadas a partir del modelo.
int getAutoResizeMode()	Obtiene el modo de cambio de tamaño automático de la tabla.
TableCellEditor getCellEditor()	Obtiene el cellEditor.

Método	Descripción
TableCellEditor getCellEditor(int row, int column)	Obtiene el editor adecuado para la celda dada por su columna y fila.
Rectangle getCellRect(int row, int column, boolean includeSpacing)	Obtiene un rectángulo que localiza la celda que reside en la intersección de fila y columna.
TableCellRenderer getCellRenderer(int row, int column)	Obtiene un renderizador adecuado para la celda dada su fila y columna.
boolean getCellSelectionEnabled()	Devuelve true si se permiten selecciones simultáneas de fila y columna.
TableColumn getColumn(Object identifier)	Obtiene el objeto TableColumn para la columna en la tabla cuyo indicador sea igual a identifier, cuando se compara utilizando equals().
Class getColumnClass(int column)	Obtiene el tipo de columna en una posición de la vista dada.
int getColumnCount()	Obtiene el número de columnas del modelo de columna (observe que éste puede ser distinto del número de columnas en el modelo de la tabla).
TableColumnModel getColumnModel()	Obtiene el TableColumnModel que contiene toda la información de columnas de esta tabla.
String getColumnName(int column)	Obtiene el nombre de la columna en una posición de vista dada.
boolean getColumnSelectionAllowed()	Devuelve true si se pueden seleccionar columnas.
TableCellEditor getDefaultEditor(Class columnClass)	Obtiene el editor que se debe utilizar cuando no se ha confi-

Método	Descripción
	gurado ningún editor en un TableColumn.
TableCellRenderer getDefaultRenderer (Class columnClass)	Obtiene el renderizador que se debe utilizar cuando no se ha seleccionado un renderizador en un TableColumn.
int getEditingColumn()	Obtiene el índice de la columna en edición.
int getEditingRow()	Obtiene el índice de la fila en edición.
Component getEditorComponent()	Obtiene el componente que devolvió el CellEditor.
Color getGridColor()	Obtiene el color utilizado para dibujar las líneas de rejilla.
Dimension getIntercellSpacing()	Obtiene el espaciamiento vertical y horizontal entre celdas.
TableModel getModel()	Obtiene el TableModel que proporciona los datos mostrados por el receptor.
Dimension getPreferredScrollableViewport-Size()	Obtiene el tamaño preferido del Viewport para esta tabla.
int getRowCount()	Obtiene el número de filas en la tabla.
int getRowHeight()	Obtiene la altura de una fila de la tabla en el receptor.
int getRowMargin()	Obtiene la cantidad de espacio libre entre filas.
boolean getRowSelectionAllowed()	Devuelve true si se pueden seleccionar filas.
int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)	Obtiene el visibleRect.height o visibleRect.width, dependiendo de la orientación de la tabla.
boolean getScrollableTracksViewportHeight()	Devuelve true si la altura del Viewport no determina la altura de la tabla.

Método	Descripción
boolean getScrollableTracksViewportWidth()	Devuelve true si la anchura del Viewport no determina la anchura de la tabla.
int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)	Obtiene el incremento de desplazamiento que expone completamente una nueva fila o columna.
int getSelectedColumn()	Obtiene el índice de la primera columna seleccionada ó -1 si no hay columnas seleccionadas.
int getSelectedColumnCount()	Obtiene el número de columnas seleccionadas.
int[] getSelectedColumns()	Obtiene los índices de todas las columnas seleccionadas.
int getSelectedRow()	Obtiene el índice de la primera fila seleccionada ó -1 si no existen filas seleccionadas.
int getSelectedRowCount()	Obtiene el número de filas seleccionadas.
int[] getSelectedRows()	Obtiene los índices de todas las filas seleccionadas.
Color getSelectionBackground()	Obtiene el color de fondo para las celdas seleccionadas.
Color getSelectionForeground()	Obtiene el color de primer plano para las celdas seleccionadas.
ListSelectionModel getSelectionModel()	Obtiene el ListSelectionModel que se utiliza para mantener el estado de selección de fila.
boolean getShowHorizontalLines()	Devuelve true si el receptor dibuja líneas horizontales entra las celdas y false si no es así.
boolean getShowVerticalLines()	Devuelve true si el receptor dibuja líneas verticales entre las celdas y false si no es así.

Método	Descripción
JTableHeader getTableHeader()	Obtiene el tableHeader que trabaja con este JTable.
String getToolTipText(MouseEvent event)	Sobrescribe el método setToolTipText de JComponent y permite el uso de ayudas de herramientas del renderizador (si el renderizador tiene texto asignado).
TableUI getUI()	Obtiene el objeto apariencia que renderiza este componente.
String getUIClassID()	Obtiene el nombre de la apariencia que renderiza este componente.
Object getValueAt(int row, int column)	Obtiene el valor de la celda en row y column.
protected void initializeLocalVars()	Inicia las propiedades de tabla con sus valores predeterminados.
boolean isCellEditable(int row, int column)	Devuelve true si la celda en row y column puede editarse.
boolean isCellSelected(int row, int column)	Devuelve true si la celda en una posición dada está seleccionada.
boolean isColumnSelected(int column)	Devuelve true si la columna en un índice dado esta seleccionada.
boolean isEditing()	Devuelve true si la tabla está editando una celda.
boolean isManagingFocus()	Sobrescrito para devolver true.
boolean isRowSelected(int row)	Devuelve true si la fila en una posición dada está seleccionada.
void moveColumn(int column, int targetColumn)	Mueve la columna column a la posición ocupada actual-

Método	Descripción
	mente por la columna target-Column.
protected String paramString()	Obtiene una representación de cadena de esta JTable.
Component prepareEditor(TableCellEditor editor, int row, int column)	Prepara el editor dado utilizando el valor de la celda dada.
Component prepareRenderer(TableCell-Renderer renderer, int row, int column)	Prepara el renderizador dado con un valor adecuado del dataModel.
void removeColumn(TableColumnaColumn)	Elimina una columna de la matriz de columnas del JTable.
void removeColumnSelectionInterval(int index0, int index1)	Deselecciona las columnas desde index0 a index1, inclusive.
void removeEditor()	Descarta el objeto editor.
void removeRowSelectionInterval(intindex0, int index1)	Elimina la selección desde index0 a index1, para filas, inclusive.
void reshape(int x, int y, int width, int height)	Llama a super.reshape() y está sobrescrito para detectar cambios en los límites de la tabla.
protected void resizeAndRepaint()	Equivalente a revalidate() seguido por repaint().
int rowAtPoint(Point point)	Obtiene el índice de la fila sobre el cual se sitúa el punto point.
void selectAll()	Selecciona todas las columnas, filas y celdas en la tabla.
void setAutoCreateColumnsFromModel (boolean createColumns)	Asigna el indicador autoCreateColumnsFromModel de la tabla.
void setAutoResizeMode(int mode)	Asigna el modo de cambio de tamaño automático de la tabla

Método	Descripción
	cuando ésta cambia de tamaño.
void setCellEditor(TableCellEditor anEditor)	Asigna la variable cellEditor.
void setCellSelectionEnabled(boolean flag)	Asigna si esta tabla permitirá tanto selección de filas como de columnas al mismo tiempo.
void setColumnModel(TableColumnModel newModel)	Asigna el modelo de columna para esta tabla a newModel y registra receptores de notificación para el nuevo modelo de columna.
void setColumnSelectionAllowed(boolean flag)	Asigna que columnas de este modelo se puedan seleccionar.
void setColumnSelectionInterval(int index0, int index1)	Selecciona las columnas desde index0 a index1 incluido.
void setDefaultEditor(Class columnClass, TableCellEditor editor)	Asigna el editor predeterminado que debe utilizarse si no se asigna un editor a un TableColumn.
void setDefaultRenderer(Class columnClass, TableCellRenderer renderer)	Asigna un renderizador predeterminado que se utilizará si no se asigna un renderizador a TableColumn.
void setEditingColumn(int aColumn)	Asigna la variable editingColumn.
void setEditingRow(int aRow)	Asigna la variable editingRow.
void setGridColor(Color newColor)	Asigna el color utilizado para dibujar las líneas de rejilla a newColor y visualiza de nuevo el receptor.
void setIntercellSpacing(Dimension newSpacing)	Asigna la anchura y altura entre celdas a newSpacing y dibuja de nuevo el receptor.
void setModel(TableModel newModel)	Asigna el modelo de datos para esta tabla a newModel y registra los receptores de

Método	Descripción
	modificaciones para el nuevo modelo de datos.
void setPreferredScrollableViewportSize (Dimension size)	Asigna el tamaño preferido del Viewport para esta tabla.
void setRowHeight(int newHeight)	Asigna la altura de filas.
void setRowMargin(int rowMargin)	Asigna la cantidad de espacio libre entre filas.
void setRowSelectionAllowed(boolean flag)	Asigna si las filas de este modelo se pueden seleccionar.
void setRowSelectionInterval(int index0, int index1)	Selecciona las filas desde index0 a index1 inclusive.
void setSelectionBackground(Color selectionBackground)	Asigna el color de fondo para las celdas seleccionadas.
void setSelectionForeground(Color selectionForeground)	Asigna el color de primer plano para las celdas seleccionadas.
void setSelectionMode(int selectionMode)	Asigna el modo de selección de tabla para permitir selección simple, un intervalo simple continuo o intervalos múltiples.
void setSelectionModel(ListSelectionModel newModel)	Asigna el modelo de selección de filas para esta tabla a newModel.
void setShowGrid(boolean b)	Asigna si se dibujan líneas de rejilla alrededor de las celdas.
void setShowHorizontalLines(boolean b)	Asigna si se dibujan líneas horizontales entre celdas.
void setShowVerticalLines(boolean b)	Asigna si se dibujan líneas verticales entre celdas.
void setTableHeader(JTableHeader newHeader)	Asigna el tableHeader que trabaja con esta tabla a newHeader.
void setUI(TableUI ui)	Asigna el objeto apariencia que renderiza este componente.

Método	Descripción
void setValueAt(Object aValue, int row, int column)	Asigna el valor para la celda en row y column.
void sizeColumnsToFit(boolean lastColumn Only)	Obsoleto. Reemplazado por sizeColumnsToFit(int).
void sizeColumnsToFit(int resizingColumn)	Cambia el tamaño de una o más columnas en la tabla para que la anchura total de todas las columnas del JTable sea igual a la anchura de la tabla.
void tableChanged(TableModelEvent e)	Llamado cuando se cambia la tabla.
void updateUI()	Llamado por UIManager cuando cambia la apariencia.
void valueChanged(ListSelectionEvent e)	Invocado cuando cambia la selección.

El modelo utilizado para contener los datos en una tabla deriva de la clase AbstractTableModel.

El modelo de tabla predeterminado es la clase DefaultTableModel. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-javax.swing.table.AbstractTableModel
        |
        +-javax.swing.table.DefaultTableModel
```

La tabla 18.4 muestra los campos de la clase DefaultTableModel, la tabla 18.5 muestra sus constructores y la tabla 18.6 sus métodos.

Tabla 18.4. Campos de la clase DefaultTableModel.

Campo	Descripción
protected Vector columnIdentifiers	Un vector de identificadores de columna.
protected Vector dataVector	Un vector de valores Object.

Tabla 18.5. Constructores de la clase DefaultTableModel.

Constructor	Descripción
DefaultTableModel()	Construye un DefaultTableModel.
DefaultTableModel(int numRows, int numColumns)	Construye un DefaultTableModel con numRows y numColumns.
DefaultTableModel(Object[][] data, Object[] columnNames)	Construye un DefaultTableModel e inicia la tabla pasando data y columnNames al método setDataVector().
DefaultTableModel(Object[] columnNames, int numRows)	Construye un DefaultTableModel que tiene tantas columnas como elementos o valores nulos de objetos en columnNames y numRows.
DefaultTableModel(Vector columnNames, int numRows)	Construye un DefaultTableModel con tantas columnas como elementos existan o valores de objeto nulos en columnNames y numRows.
DefaultTableModel(Vector data, Vector columnNames)	Construye un DefaultTableModel e inicia la tabla pasando data y columnNames al método setDataVector().

Tabla 18.6. Métodos de la clase DefaultTableModel.

Método	Descripción
void addColumn(Object columnName)	Añade una columna al modelo.
void addColumn(Object columnName, Object[] columnData)	Añade una columna al modelo con el nombre columnName.
void addColumn(Object columnName, Vector columnData)	Añade una columna al modelo.
void addRow(Object[] rowData)	Añade una fila al final del modelo.
void addRow(Vector rowData)	Añade la fila al final del modelo.

Método	Descripción
protected static Vector convertToVector (Object[] anArray)	Obtiene un Vector que contiene los mismos objetos que la matriz.
protected static Vector convertToVector (Object[][] anArray)	Obtiene un Vector de Vectores que contiene los mismos objetos que la matriz.
int getColumnCount()	Obtiene el número de columnas en esta tabla de datos.
String getColumnName(int column)	Obtiene el nombre de columna.
Vector getDataVector()	Devuelve el vector de Vectores que contiene los valores de datos de la tabla.
int getRowCount()	Obtiene el número de filas en esta tabla de datos.
Object getValueAt(int row, int column)	Obtiene un valor de atributo para la celda en row y column.
void insertRow(int row, Object[] rowData)	Inserta una fila en row en el modelo.
void insertRow(int row, Vector rowData)	Inserta una fila en row en el modelo.
boolean isCellEditable(int row, int column)	Devuelve true si la celda en row y column se puede editar.
void moveRow(int startIndex, int endIndex, int toIndex)	Mueve una o más filas comenzando desde startIndex hasta endIndex en el modelo hasta toIndex.
void newDataAvailable(TableModelEvent event)	Equivalente a fireTableChanged.
void newRowsAdded(TableModelEvent event)	Este método permite asegurar que las nuevas filas tienen el número correcto de columnas.
void removeRow(int row)	Elimina la fila en row del modelo.
void rowsRemoved(TableModelEvent event)	Equivalente a fireTableChanged().

Método	Descripción
void setColumnIdentifiers(Object[] newIdentifiers)	Reemplaza los identificadores de columna del modelo.
void setColumnIdentifiers(Vector newIdentifiers)	Reemplaza los identificadores de columna del modelo.
void setDataVector(Object[][] newData, Object[] columnNames)	Reemplaza el valor de la variable de instancia dataVector con los valores de la matriz newData.
void setDataVector(Vector newData, Vector columnNames)	Reemplaza la invariable de instancia actual dataVector con los nuevos rectores de filas, newData.
void setNumRows(int newSize)	Asigna el número de filas del modelo.
void setValueAt(Object aValue, int row, int column)	Asigna el valor del objeto para row y column.

Aunque la clase JTable es compleja, proporciona valores predeterminados para la mayoría de los aspectos de la programación de tablas, facilitando las cosas. Aquí tiene un ejemplo en el que utilizamos la clase DefaultTableModel para añadir datos a una tabla, fila por fila. Primero, creamos un nuevo objeto de la clase DefaultTableModel y un nuevo objeto JTable que visualiza los datos en ese modelo de objeto:

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

    /*
<APPLET
    CODE = table.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class table extends JApplet
{
    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);
    ...
}
```

A continuación, utilizamos el método addColumn del modelo de tabla predeterminado para añadir columnas a la tabla, proporcionando los títulos Columna 0, Columna 1 y así sucesivamente:

```
public class table extends JApplet
{
    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()
    {
        for(int column = 0; column < 5; column++){
            defaulttablemodel.addColumn("Columna " + column);
        }
    ...
}
```

Una vez añadidas las columnas a la tabla, estamos listos para añadir los datos que la tabla mostrará en sus filas y lo haremos construyendo cada fila y después utilizando el método addRow del modelo. Cada fila es una matriz de objetos; en este caso, utilizaremos objetos String. Aquí tiene el mecanismo de construcción de cada fila y la forma de añadirla a la tabla:

```
public class table extends JApplet
{
    Object[] data = new Object[5];

    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()
    {
        for(int column = 0; column < 5; column++){
            defaulttablemodel.addColumn("Columna " + column);
        }

        for(int row = 0; row < 10; row++) {
            for(int column = 0; column < 5; column++) {
                data[column] = "Celda " + row + "," + column;
            }
            defaulttablemodel.addRow(data);
        }
        ...
    }
}
```

Lo único que resta es visualizar la nueva tabla. Las tablas normalmente se muestran en paneles de desplazamiento y, de hecho, debería hacerlo de esa forma para estar seguro de que aparecen las cabeceras de columna. Aquí tiene la forma de añadir la tabla a un JScrollPane y después mostrarla:

```
public class table extends JApplet
{
    Object[] data = new Object[5];

    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()
    {
        for(int column = 0; column < 5; column++){
            defaulttablemodel.addColumn("Columna " + column);
        }

        for(int row = 0; row < 10; row++) {
            for(int column = 0; column < 5; column++) {
                data[column] = "Celda " + row + "," + column;
            }
            defaulttablemodel.addRow(data);
        }
        getContentPane().add(new JScrollPane(jtable));
    }
}
```

Crear árboles

"Oh", dice el programador novato, "el gran jefe quiere que cree una utilidad de directorio de archivos que muestre archivos y directorios utilizando un árbol". "Bien", decimos, "puede utilizar un *árbol*". "Fantástico", dice el programador novato. "Una cosa, ¿qué es un árbol?"

Los árboles son controles que muestran datos jerárquicos como un esquema basado en nodos. Un nodo específico puede quedar identificado bien por un TreePath (un objeto que encapsula un nodo y todos los nodos de que desciende; llamados *ancestros*) o por su fila visualizada (la fila en pantalla muestra únicamente un nodo). Puede expandir nodos para visualizar todos sus hijos. Uno *nodo colapsado* es el que oculta sus hijos y un *nodo oculto* es el que queda colapsado por su padre. Los árboles están soportados en la Swing por la clase JTree. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.JTree
```

La tabla 18.7 muestra los campos de la clase JTree, la tabla 18.8 muestra sus constructores y la tabla 18.9 sus métodos.

Tabla 18.7. Campos de la clase JTree.

Campo	Descripción
static String CELL_EDITOR_PROPERTY	El nombre de la propiedad ligada para cellEditor.
static String CELL_RENDERER_PROPERTY	El nombre de la propiedad ligada para cellRenderer.
protected TreeCellEditor cellEditor	El editor para las entradas.
protected TreeCellRenderer cellRenderer	El renderizador de celda utilizado para dibujar los nodos.
protected boolean editable	Devuelve true si el árbol es editable.
static String EDITABLE_PROPERTY	El nombre de la propiedad ligada para editable.
static String INVOKES_STOP_CELL_ EDITING_PROPERTY	El nombre de la propiedad ligada para messagesStopCellEditing.
protected boolean invokesStopCellEditing	Devuelve true cuando se detiene la edición.
static String LARGE_MODEL_PROPERTY	El nombre de la propiedad ligada para largeModel.
protected boolean largeModel	Devuelve true si el árbol utiliza modelo largo.
static String ROOT_VISIBLE_PROPERTY	El nombre la propiedad ligada para rootVisible.
protected boolean rootVisible	Devuelve true si el nodo raíz se visualiza y false si sus hijos son los nodos visibles superiores.
static String ROW_HEIGHT_PROPERTY	El nombre de la propiedad ligada para rowHeight.
protected int rowHeight	La altura utilizada por cada fila visible.

Campo	Descripción
static String SCROLLS_ON_EXPAND_PROPERTY	El nombre de la propiedad ligada para scrollsOnExpand.
protected boolean scrollsOnExpand	Devuelve true si un nodo se desplaza cuando es expandido para mostrar todos sus descendientes.
static String SELECTION_MODEL_PROPERTY	El nombre de la propiedad ligada para selectionModel.
protected TreeSelectionModel selectionModel	Modeliza el conjunto de nodos seleccionados en este árbol.
protected JTree.TreeSelectionRedirector selectionRedirector	Crea un nuevo evento y lo pasa a selectionListeners.
static String SHOWS_ROOT_HANDLES_PROPERTY	El nombre de la propiedad ligada para showsRootHandles.
protected boolean showsRootHandles	Devuelve true si se muestran los sectores de primer nivel del árbol.
protected int toggleClickCount	El número de clics del ratón antes de expandir un nodo.
static String TREE_MODEL_PROPERTY	El nombre la propiedad ligada para treeModel.
protected TreeModel treeModel	El modelo que define el árbol visualizado para este objeto.
protected TreeModelListener treeModelListener	Actualiza expandedState.
static String VISIBLE_ROW_COUNT_PROPERTY	El nombre la propiedad ligada para visibleRowCount.
protected int visibleRowCount	El número de filas visibles cada vez.

Tabla 18.8. Constructores de la clase JTree.

Constructor	Descripción
JTree()	Construye un JTree con un modelo de ejemplo.

Constructor	Descripción
JTree(Hashtable value)	Construye un JTree creado a partir de una tabla *hash*.
JTree(Object[] value)	Construye un JTree con cada elemento de la matriz dada como hijo de un nuevo nodo raíz.
JTree(TreeModel newModel)	Construye una instancia de JTree que visualiza un nodo raíz utilizando el modelo de datos proporcionado.
JTree(TreeNode root)	Construye un JTree con el TreeNode dado como raíz, que muestra el nodo raíz.
JTree(TreeNode root, boolean asksAllowsChildren)	Construye un JTree con el TreeNode dado como raíz, que visualiza el nodo raíz y decide si un nodo es un nodo hoja en la manera indicada.
JTree(Vector value)	Construye un JTree con cada elemento del Vector dado como hijo de un nuevo nodo raíz que no se visualiza.

Tabla 18.9. Métodos de la clase JTree.

Método	Descripción
void addSelectionInterval(int index0, int index1)	Añade a la selección los caminos entre index0 y index1, incluido.
void addSelectionPath(TreePath path)	Añade el nodo identificado por el TreePath dado a la selección.
void addSelectionPaths(TreePath[] paths)	Añade cada camino en la matriz de caminos a la selección.
void addSelectionRow(int row)	Añade el camino de la fila actual a la selección.
void addSelectionRows(int[] rows)	Añade el camino de cada una de las filas dadas a la selección.

Método	Descripción
void addTreeExpansionListener(TreeExpansionListener tel)	Añade un receptor para eventos TreeExpansion.
void addTreeSelectionListener(TreeSelectionListener tsl)	Añade un receptor para eventos TreeSelection.
void addTreeWillExpandListener(TreeWillExpandListener tel)	Añade un receptor para eventos TreeWillExpand.
void cancelEditing()	Cancela la sesión de edición actual.
void clearSelection()	Limpia la selección.
protected void clearToggledPaths()	Limpia el caché de caminos alternativos.
void collapsePath(TreePath path)	Asegura que el nodo identificado por el camino actual está colapsado y es visible.
void collapseRow(int row)	Asegura que el nodo en la fila especificada está colapsado.
String convertValueToText(Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)	Llamada por los renderizadores para convertir el valor dado al texto.
protected static TreeModel createTreeModel(Object value)	Obtiene un TreeModel que encapsula el objeto dado.
protected TreeModelListener createTreeModelListener()	Crea y devuelve una instancia de TreeModelHandler.
void expandPath(TreePath path)	Asegura que el nodo identificado por el camino dado se expande.
void expandRow(int row)	Asegura que el nodo de la fila dada está expandido.
void fireTreeCollapsed(TreePath path)	Notifica a todos los receptores sobre este evento.
void fireTreeExpanded(TreePath path)	Notifica a todos los receptores sobre este evento.
void fireTreeWillCollapse(TreePath path)	Notifica a todos los receptores sobre este evento.

Método	Descripción
void fireTreeWillExpand(TreePath path)	Notifica a todos los receptores sobre este evento.
protected void fireValueChanged(TreeSelectionEvent e)	Dispara un evento "valor cambiado".
AccessibleContext getAccessibleContext()	Obtiene el AccessibleContext.
TreeCellEditor getCellEditor()	Obtiene el editor utilizado para editar entradas.
TreeCellRenderer getCellRenderer()	Obtiene el TreeCellRenderer que renderiza cada celda.
TreePath getClosestPathForLocation(int x, int y)	Obtiene el camino al nodo más cercano a x, y.
int getClosestRowForLocation(int x, int y)	Obtiene la fila del nodo más cercano a x, y.
protected static TreeModel getDefaultTreeModel()	Crea y devuelve un TreeModel de ejemplo.
protected Enumeration getDescendantToggledPaths(TreePath parent)	Obtiene una enumeración de caminos TreePaths que ha sido expandida y desciende de parent.
TreePath getEditingPath()	Obtiene el camino del elemento que se edita en este momento.
Enumeration getExpandedDescendants (TreePath parent)	Obtiene una enumeración de los descendientes de path que están expandidos ahora.
boolean getInvokesStopCellEditing()	Obtiene el indicador que informa de lo que sucede cuando se interrumpe la edición.
Object getLastSelectedPathComponent()	Obtiene el último componente camino en el primer nodo de la selección actual.
TreePath getLeadSelectionPath()	Obtiene el camino al último nodo añadido a la selección actual.
int getLeadSelectionRow()	Obtiene el índice de fila del último nodo añadido a la selección.

Método	Descripción
int getMaxSelectionRow()	Obtiene la última fila seleccionada.
int getMinSelectionRow()	Obtiene la primera fila seleccionada.
TreeModel getModel()	Obtiene el TreeModel que proporciona los datos.
protected TreePath[] getPathBetweenRows(int index0, int index1)	Obtiene instancias JTreePath que representan el camino entre index0 e index1, incluido.
Rectangle getPathBounds(TreePath path)	Obtiene el rectángulo en el que se dibuja el nodo dado.
TreePath getPathForLocation(int x, int y)	Obtiene el camino para el nodo en una posición dada.
TreePath getPathForRow(int row)	Obtiene el camino para la fila dada.
Dimension getPreferredScrollableViewportSize()	Obtiene el tamaño visual preferido de un JTree.
Rectangle getRowBounds(int row)	Obtiene el rectángulo en que se dibuja el nodo en una fila dada.
int getRowCount()	Obtiene el número total de filas.
int getRowForLocation(int x, int y)	Obtiene la fila en la posición dada.
int getRowForPath(TreePath path)	Obtiene la fila que visualiza el nodo identificado por el camino dado.
int getRowHeight()	Obtiene la altura de cada fila.
int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)	Obtiene el incremento de bloque.
boolean getScrollableTracksViewportHeight()	False indica que la altura del Viewport no determina la altura de la tabla.
boolean getScrollableTracksViewportWidth()	False indica que la anchura del Viewport no determina la anchura de la tabla.

Método	Descripción
int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)	Obtiene la cantidad de incremento en el desplazamiento.
boolean getScrollsOnExpand()	Devuelve true si el árbol se desplaza para mostrar los nodos escondidos previamente.
int getSelectionCount()	Obtiene el número de nodos seleccionados.
TreeSelectionModel getSelectionModel()	Obtiene el modelo para las selecciones.
TreePath getSelectionPath()	Obtiene el camino del primer nodo seleccionado.
TreePath[] getSelectionPaths()	Obtiene el camino de todos los valores seleccionados.
int[] getSelectionRows()	Obtiene todas las filas seleccionadas en curso.
boolean getShowsRootHandles()	Devuelve true si se muestran los iconos para los nodos raíz.
String getToolTipText(MouseEvent event)	Sobrescribe el método getToolTipText de JComponent para permitir que se muestren las ayudas de herramientas.
TreeUI getUI()	Obtiene el objeto apariencia que dibuja este componente.
String getUIClassID()	Obtiene el nombre de la clase apariencia que dibuja este componente.
int getVisibleRowCount()	Obtiene el número de filas que se visualiza en el área disponible.
boolean hasBeenExpanded(TreePath path)	Devuelve true si el nodo identificado por el camino se ha expandido alguna vez.
boolean isCollapsed(int row)	Devuelve true si el nodo en la posición dada de la fila está colapsado.

Método	Descripción
boolean isCollapsed(TreePath path)	Devuelve true si el valor identificado por path está actualmente colapsado.
boolean isEditable()	Devuelve true si el árbol puede ser editado.
boolean isEditing()	Devuelve true si el árbol está en modo de edición.
boolean isExpanded(int row)	Devuelve true si el nodo en una fila dada de pantalla está actualmente expandido.
boolean isExpanded(TreePath path)	Devuelve true si el nodo identificado por el camino está actualmente expandido.
boolean isFixedRowHeight()	Devuelve true si la altura de cada fila en pantalla tiene tamaño fijo.
boolean isLargeModel()	Devuelve true si el árbol está configurado para modelo largo.
boolean isPathEditable(TreePath path)	Obtiene el valor de isEditable.
boolean isPathSelected(TreePath path)	Devuelve true si el elemento identificado por el camino está seleccionado actualmente.
boolean isRootVisible()	Devuelve true si el nodo raíz del árbol es visible.
boolean isRowSelected(int row)	Devuelve true si el nodo identificado por row está seleccionado.
boolean isSelectionEmpty()	Devuelve true si la selección está actualmente vacía.
boolean isVisible(TreePath path)	Devuelve true si el valor identificado por path es visible actualmente.
void makeVisible(TreePath path)	Asegura que el nodo identificado por path es actualmente visible.

Método	Descripción
protected String paramString()	Obtiene una representación de cadena de este JTree.
protected void removeDescendantToggledPaths(Enumeration toRemove)	Elimina cualquier descendiente de toRemove en TreePaths que siga expandido.
void removeSelectionInterval(int index0, int index1)	Elimina de la selección los nodos entre index0 e index1, incluido.
void removeSelectionPath(TreePath path)	Elimina el nodo identificado por el camino dado de la selección.
void removeSelectionPaths(TreePath[] paths)	Elimina el nodo identificado por los caminos dados de la selección actual.
void removeSelectionRow(int row)	Elimina el camino en el índice de fila de la selección actual.
void removeSelectionRows(int[] rows)	Elimina los caminos seleccionados en cada una de las filas dadas.
void removeTreeExpansionListener(TreeExpansionListener tel)	Elimina un receptor de eventos TreeExpansion.
void removeTreeSelectionListener(TreeSelectionListener tsl)	Elimina un receptor TreeSelection.
void removeTreeWillExpandListener(TreeWillExpandListener tel)	Elimina al receptor de eventos TreeWillExpand.
void scrollPathToVisible(TreePath path)	Se desplaza para que el nodo identificado por path sea visible.
void scrollRowToVisible(int row)	Desplaza el elemento identificado por row hasta que sea visible.
void setCellEditor(TreeCellEditor cellEditor)	Asigna el editor de celdas.
void setCellRenderer(TreeCellRenderer x)	Asigna el TreeCellRenderer que se utilizará para dibujar cada celda.
void setEditable(boolean flag)	Determina si el árbol es editable.

Método	Descripción
protected void setExpandedState (TreePath path, boolean state)	Asigna el estado expandido del receptor.
void setInvokesStopCellEditing(boolean newValue)	Determina lo que sucede cuando se interrumpe la edición.
void setLargeModel(boolean newValue)	Especifica si la interfaz de usuario debería utilizar un modelo largo.
void setModel(TreeModel newModel)	Asigna el TreeModel que proporciona los datos.
void setRootVisible(boolean rootVisible)	Determina si el nodo raíz de TreeModel es visible.
void setRowHeight(int rowHeight)	Asigna la altura de cada celda.
void setScrollsOnExpand(boolean newValue)	Determina si algo debería desplazarse cuando se expande un nodo.
void setSelectionInterval(int index0, int index1)	Selecciona los nodos entre index0 e index1, incluido.
void setSelectionModel(TreeSelectionModel selectionModel)	Asigna el modelo de selección del árbol.
void setSelectionPath(TreePath path)	Selecciona el nodo identificado por un camino dado.
void setSelectionPaths(TreePath[] paths)	Selecciona los nodos identificados por una matriz de caminos.
void setSelectionRow(int row)	Selecciona el nodo en la fila dada en pantalla.
void setSelectionRows(int[] rows)	Selecciona los nodos correspondientes a cada una de las filas dadas en pantalla.
void setShowsRootHandles(boolean newValue)	Determina si los iconos de nodo se visualizan.
void setUI(TreeUI ui)	Asigna el objeto apariencia que renderiza este componente.

Método	Descripción
void setVisibleRowCount(int newCount)	Asigna el número de filas visibles.
void startEditingAtPath(TreePath path)	Selecciona el camino dado e inicia la edición.
boolean stopEditing()	Finaliza la sesión de edición.
void treeDidChange()	Se llama cuando el árbol ha cambiado lo suficiente y necesita cambiar sus límites de tamaño.
void updateUI()	Llamado por UIManager cuando la apariencia cambia.

Puede añadir datos ahora a un árbol de diversas formas; en este capítulo, utilizaremos una técnica conocida: la creación de una tabla *hash*, rellenar los datos y añadir esa *hash* al árbol. Puede utilizar JTree para mostrar nodos compuestos (por ejemplo, nodos que contienen tanto un icono gráfico como texto), subclasificando TreeCellRenderer y utilizando setTreeCellRenderer. Para editar nodos, puede subclasificar TreeCellEditor y utilizar setTreeCellEditor.

Aquí tiene un ejemplo rápido. En este caso, basta crear un árbol predeterminado y añadirlo a un panel de desplazamiento:

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

    /*
<APPLET
    CODE = tree.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class tree extends JApplet
{
    public void init()
    {
        JTree jtree = new JTree();

        getContentPane().add(new JScrollPane(jtree));
    }
}
```

Este código es todo lo que necesita para mostrar un árbol básico; la figura 18.1 muestra este código, donde puede ver que la Swing ha añadido algunos datos predeterminados al árbol. Este ejemplo se encuentra en el archivo tree.java del CD.

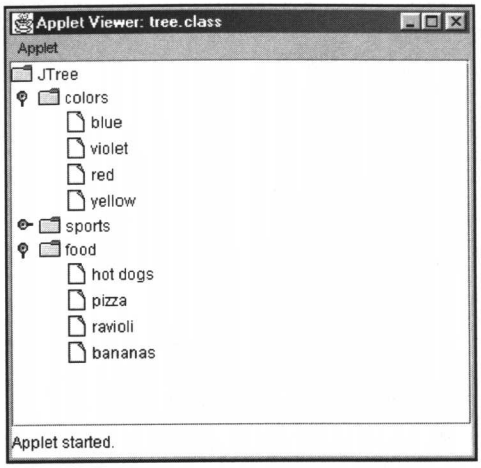


Figura 18.1. Un árbol básico.

19 Swing: Componentes de texto

Crear componentes de texto en Swing: la clase JTextComponent

Los componentes de texto de la Swing se crean sobre la clase JTextComponent y merece la pena examinar los campos, constructores y métodos de esta clase debido a que todos los componentes de la Swing heredan de él. Aquí tiene el diagrama de herencia para la clase JTextComponent:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.text.JTextComponent
```

Crear campos de texto

El campo de texto es el control de texto básico de la Swing. De hecho, ya hemos utilizado campos de texto previamente en el capítulo 12. Aquí tiene el diagrama de herencia de la clase JTextField:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.text.JTextComponent
                          |
                          +--javax.swing.JTextField
```

En el capítulo 12, encontrará los constructores de la clase JTextField mostrados en la tabla 12.6 y sus métodos en la tabla 12.7. Aquí tiene un breve ejemplo que hace funcionar un campo de texto:

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

    /*
<APPLET
    CODE = textfield.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class textfield extends JApplet
{
    JTextField text = new JTextField(20);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(text);
        text.setText ( "¡Hola desde Swing!" );
    }
}
```

Examinaremos alguna de las capacidades de los campos de texto de la Swing en los dos temas siguientes.

Ajustar la alineación del campo de texto

"Mmh", dice el programador novato, "me ha llegado un memorándum extraño del gran jefe. Dice: "Estamos probando algo nuevo. Alinear a la

izquierda todo el texto". ¿Es algo político?". Sonreímos y decimos, "No, se trata de los campos de texto".

Puede asignar la alineación de los campos de texto con el método setHorizontalAlignment (no existe método setVerticalAlignment). Aquí tiene un ejemplo que muestra las posibilidades: derecha, izquierda y centrada. En este caso, permitimos al usuario hacer clic sobre un botón para asignar la alineación del texto en un campo de texto; después, utilizamos el método setHorizontalAlignment para justificar el texto y que coincida. Aquí tiene el código:

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

    /*
<APPLET
    CODE = textfieldalign.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class textfieldalign extends JApplet
{
    JTextField jtextfield = new JTextField("¡Hola desde Swing!");
    JButton jbutton1, jbutton2, jbutton3;

    public void init()
    {
        Container contentPane = getContentPane();

        jtextfield.setColumns(30);

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new buttonpanel());
        contentPane.add(jtextfield);
    }

    class buttonpanel extends JPanel implements ActionListener
    {

        public buttonpanel()
        {

            jbutton1 = new JButton("Izquierda");
            jbutton2 = new JButton("Derecha");
            jbutton3 = new JButton("Centrado");

            jbutton1.addActionListener(this);
            jbutton2.addActionListener(this);
            jbutton3.addActionListener(this);
```

```
            setLayout(new FlowLayout());

            add(new JLabel("Justificación"));
            add(jbutton1);
            add(jbutton2);
            add(jbutton3);

        }

        public void actionPerformed(ActionEvent e)
        {
            if(e.getSource() == jbutton1) {
                jtextfield.setHorizontalAlignment(JTextField.LEFT);
            } else if(e.getSource() == jbutton2) {
                jtextfield.setHorizontalAlignment(JTextField.RIGHT);
            } else if(e.getSource() == jbutton3) {
                jtextfield.setHorizontalAlignment(JTextField.CENTER);
            }
        }
    }
}
```

La figura 19.1 muestra el resultado de este código, donde el usuario puede asignar la alineación del texto en el campo de texto con sólo hacer clic sobre un botón. Este ejemplo se encuentra en el archivo textfieldalign.java del CD que acompaña a este libro.



Truco: Observe que asignamos el número de columnas utilizando setColumns en el campo de texto de este ejemplo para que pueda ver el espacio horizontal que rodea al texto actual. Existen dos cosas a observar aquí: la primera, que el tamaño de una columna de un campo de texto es la anchura de la letra m, por tanto setColumns no asigna realmente el número visible de caracteres. Segundo, setColumns asigna únicamente el tamaño preferente del campo de texto.

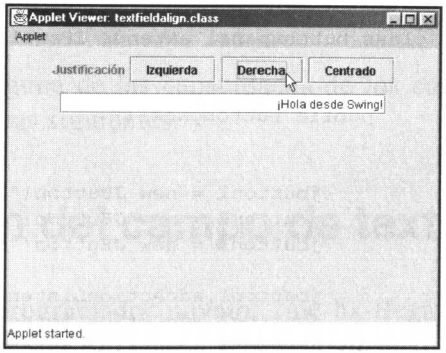


Figura 19.1. Alineación del texto de un campo de texto.

Desplazar campos de texto

"Tengo un problema", dice el programador novato. "Estoy escribiendo mi novela en un campo de texto y realmente odio tener que desplazarme por todo el texto utilizando el teclado". "Bien", decimos. "Puede desplazar el campo de texto, pero debería utilizar en su lugar un área de texto cuando trabaje con gran cantidad de texto". El PN dice: "¡Cuénteme más!"

Los campos de texto de la Swing implementan la interfaz Scrollable, lo que implica que podemos desplazar el texto bajo control de programa o dentro de paneles de desplazamiento. Aquí tiene un ejemplo que muestra cómo funciona. En este caso, utilizamos el método getHorizontalVisibility del campo de texto para obtener un objeto que implemente la interfaz BoundedRangeModel y pasamos este objeto al constructor de un deslizador, implementando directamente el desplazamiento, sin un panel de desplazamiento. Cuando el usuario ajuste el deslizador, utilizamos el método setScrollOffset del campo de texto para desplazar el texto contenido en el campo. Primero, no obstante, examine los métodos de la interfaz BoundedRangeModel, que están especialmente diseñados para facilitar los controles de desplazamiento.

Aquí tiene el código real. En este caso, situamos una cadena larga en un campo de texto corto y conectamos el campo de texto a un deslizador, como se esquematiza al comienzo de este tema:

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

    /*
<APPLET
    CODE = textfieldscroll.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
    */

public class textfieldscroll extends JApplet
{
    private JTextField jtextfield = new JTextField(
      "Aquí tenemos un bonito texto largo para un campo de texto.", 12);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new jsliderpanel());
```

```
        contentPane.add(jtextfield);
    }

    class jsliderpanel extends JPanel
    {
        JSlider jslider = new
JSlider(jtextfield.getHorizontalVisibility());

        public jsliderpanel() {
            add(new JLabel("Desplazar el texto:"));

            add(jslider);

            jslider.addChangeListener(new ChangeListener() {
                public void stateChanged(ChangeEvent e) {
                    jtextfield.setScrollOffset(jslider.getValue());
                }
            });
        }
    }
}
```

La figura 19.2 muestra el resultado. Como podemos ver, este applet contiene un campo de texto y un deslizador y podemos desplazar el texto del campo utilizando el deslizador. Eso es todo; este applet es un éxito. Encontrará el archivo textfieldscroll.java en el CD.

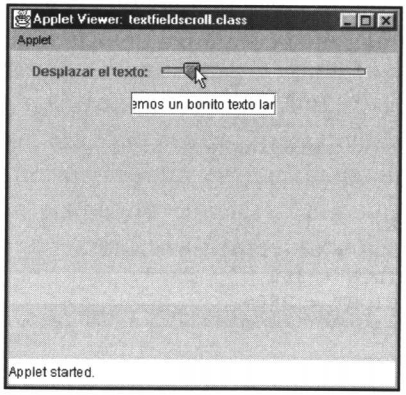


Figura 19.2. Desplazamiento de un campo de texto.

Crear campos de palabra clave

"¡Ayuda!" Llora una voz sobre el teléfono y decidimos que debe ser el programador novato. "¿Qué va mal, PN?", preguntamos. "¡El dichoso Felipe se pone detrás de mí y lee mi palabra clave mientras escribo!", dice el PN.

"Bien", decimos, "cambie sus palabras claves por otras nuevas; y pase a campos de palabra clave en lugar de campos de texto". "Gracias", dice el PN, aliviado.

La Swing incluye un control especial para la introducción de palabras clave; el control JPasswordField. Puede asignar el carácter que se muestra con cada tecla en este control cada vez que el usuario escribe un carácter, para que la palabra clave escrita no sea visible realmente. Además, la copia al portapapeles está inhabilitada para este control. De esta forma, nadie puede copiar y pegar su palabra clave. Aquí tiene el diagrama de herencia para JPasswordField:

```
java.lang.Object
  |
  +-java.awt.Component
        |
        +-java.awt.Container
              |
              +-javax.swing.JComponent
                    |
                    +-javax.swing.text.JTextComponent
                          |
                          +-javax.swing.JTextField
                                |
                                +-javax.swing.JPasswordField
```

Utilizamos el método setEchoChar para asignar el carácter de eco en un campo de palabra clave y podemos utilizar el método getPassword para leer la palabra clave escrita. El método getPassword devuelve una matriz de caracteres, pero podemos transformarla a un objeto String pasándola al constructor de String.

Aquí tiene un ejemplo en el que la palabra clave actual es "ábrete sésamo" y, si el usuario escribe esta palabra clave y pulsa **Intro**, el applet visualizará "**Correcta**" en su barra de estado; en cualquier otro caso, visualizará "**Incorrecta**". Aquí tiene el código (observe que para capturar eventos de pulsación de **Intro**, hemos añadido un receptor de acción al control de palabra clave):

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

    /*
<APPLET
    CODE = password.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
    */

public class password extends JApplet
```

```
{
    String correctpassword = "ábrete sésamo";
    JPasswordField jpasswordfield = new JPasswordField(10);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new JLabel("Introduzca su palabra clave: "));
        contentPane.add(jpasswordfield);

        jpasswordfield.setEchoChar('*');

        jpasswordfield.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
              String input = new String(jpasswordfield.getPassword());

                if(correctpassword.equals(input))
                    showStatus("Correcta");
                else
                    showStatus("Incorrecta");
            }
        });
    }
}
```

La figura 19.3 muestra el resultado de este código. Como vemos, el campo de palabra clave visualiza el carácter de eco y comprueba la palabra clave cuando el usuario pulsa **Intro**. Este ejemplo se encuentra en el archivo password.java del CD.

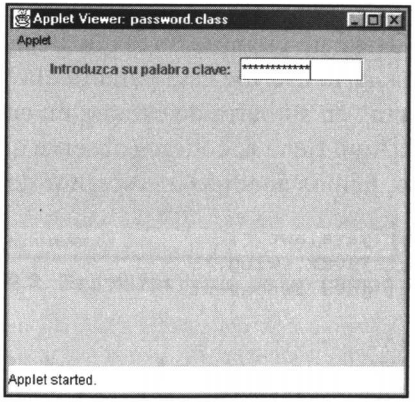


Figura 19.3. Uso de un campo de palabra clave.

20 Stream I/O y archivos

Usar la clase File

"Mmh", dice el programador novato, "es un poco molesto; mi programa intentaba abrir un archivo que no existía, justo cuando lo ejecutaba para el gran jefe". "Bien", decimos, "¿por qué no probar primero si existía el archivo utilizando el método exists de la clase File?". El PN dice, "¡Explíquemelo!"

Puede utilizar la clase File para determinar muchas cosas sobre archivos, como su tamaño, si se puede escribir sobre ellos, si un elemento es un archivo, directorio o conexión de nombres, y más aún. Aquí tiene el diagrama de herencia para la clase File:

```
java.lang.Object
  |
  +--java.io.File
```

Observe que puede utilizar isFile para determinar si un elemento es un archivo verdadero u otra cosa, como un directorio (lo puede comprobar con isDirectory) o una conexión de nombres. Utilice toURL para convertir la vía de acceso de un archivo a un URL, renombrar archivos y otros con la clase File.

Examinaremos un ejemplo: haremos funcionar varios métodos de la clase File, obteniendo alguna información sobre un archivo llamado file.txt. Aquí tiene la apariencia del código:

```
import java.io.File;

class file
{
    public static void main(String args[])
    {
        File file1 = new File("file.txt");

     System.out.println("Archivo: " + file1.getName() + (file1.isFile()?
            " es un archivo" : " es un filtro"));
        System.out.println("Tamaño: " + file1.length());
        System.out.println("Vía de acceso: " + file1.getPath());
        System.out.println("Vía de acceso absoluta: " +
file1.getAbsolutePath());
        System.out.println("Ultima modificación del archivo: "
file1.lastModified());
        System.out.println(file1.exists() ? "El archivo existe" : "El
archivo no existe");
       System.out.println(file1.canRead() ? "Se puede leer el archivo":
            "No se puede leer el archivo");
        System.out.println(file1.canWrite() ? "Se puede escribir al
archivo" :
            "No se puede leer el archivo");
        System.out.println(file1.isDirectory() ? "El archivo es un
directorio" :
            "El archivo no es un directorio");
    }
}
```

Aquí tiene los resultados de este código (observe que la hora en que se modificó por última vez el archivo se recibe de la forma habitual; ms desde el 1/1/70, que puede convertir a una fecha más legible con la clase Date):

```
java file
Archivo: file.txt es un archivo
Tamaño: 23
Vía de acceso: file.txt
Vía de acceso absoluta: C:\file.txt
Ultima modificación del archivo: 953020822000
El archivo existe
Se puede leer el archivo
Se puede escribir al archivo
El archivo no es un directorio
```

Esto es todo. Este ejemplo se encuentra en el archivo file.java del CD que acompaña a este libro. Como puede ver, el uso de la clase File puede determinar un gran impacto sobre archivos y directorios.

Trabajar con InputStream

El programador novato aparece y dice, "Estoy listo para comenzar a trabajar con archivos. ¿Dónde comienzo?" "Comience", decimos, "con las clases InputStream y OutputStream".

Las clases InputStream y OutputStream son las clases base de la entrada y salida orientadas a bytes en Java, por lo que merece la pena examinar los métodos que proporcionan estas clases a todas las demás clases orientadas a stream de bytes. Aquí tiene el diagrama de herencia para la clase InputStream, en la que se basan los streams de entrada:

```
java.lang.Object
  |
  +--java.io.InputStream
```

Examinaremos la clase OutputStream en el tema siguiente.

Trabajar con OutputStream

El opuesto de la clase InputStream, introducida en el tema previo, es la clase OutputStream, en la que se basan los streams de salida. Aquí tiene el diagrama de herencia de OutputStream:

```
java.lang.Object
  |
  +--java.io.OutputStream
```

Ahora que hemos visto InputStream y OutputStream, es el momento de hacer funcionar estas clases, comenzando en el tema siguiente.

Trabajar con FileInputStream

"Necesito abrir un archivo de imagen y trabajar con sus bytes individuales", dice el programador novato. "¿Cómo puedo hacerlo?" "Puede utilizar la clase FileInputStream para trabajar con los bytes de un archivo", decimos. El PN dice "¡Explíquemelo!"

La clase FileInputStream está diseñada especialmente para trabajar con archivos de entrada orientados a bytes y deriva de la clase InputStream, como se muestra aquí:

```
java.lang.Object
  |
```

```
+--java.io.InputStream
      |
      +--java.io.FileInputStream
```

Para crear un stream de entrada de archivo, utilizamos el constructor de la clase FileInputStream. Aquí tiene los métodos de lectura que debe utilizar con esta clase:

- ***int read()***. Lee un byte de datos desde su stream de entrada y lo devuelve.
- ***int read(byte[] b)***. Lee hasta b.length bytes de datos del stream de entrada en una matriz de bytes y devuelve el número leído de bytes.
- ***int read(byte[] b, int off, int len)***. Lee hasta len bytes de datos de este stream de entrada en una matriz de bytes y devuelve el número leído de bytes.

Vamos a examinar un ejemplo. En este caso, abriremos el propio código fuente de la aplicación, leeremos y mostraremos 50 bytes de código, saltaremos 50 bytes con el método skip y después leeremos y mostraremos 50 bytes más (observe que también podemos determinar cuántos bytes existen a la espera de lectura utilizando el método *available*, y que podemos cerrar el stream al final del código):

```
import java.io.*;

class fileinputstream
{
    public static void main(String args[]) throws Exception
    {
        int size;
        FileInputStream fileinputstream = new
FileInputStream("fileinputstream.java");

        System.out.println("Bytes disponibles: " + (size =
fileinputstream.available()));
        System.out.println("Leyendo 50 bytes....");

        byte bytearray[] = new byte[50];
        if (fileinputstream.read(bytearray) != 50) {
          System.out.println("No se pudieron leer 50 bytes");
        }

        System.out.println(new String(bytearray, 0, 50));

        System.out.println("Ignorando 50 bytes...");

        fileinputstream.skip(50);

        System.out.println("Leyendo 50 bytes....");
```

```
            if (fileinputstream.read(bytearray) != 50) {
              System.out.println("No se pudieron leer 50 bytes");
            }
            System.out.println(new String(bytearray, 0, 50));

            fileinputstream.close();
    }
}
```

Aquí tiene la salida de este código:

```
java fileinputstream
Bytes disponibles: 982
Leyendo 50 bytes....
import java.io.*;

class fileinputstream
{

Ignorando 50 bytes...
Leyendo 50 bytes....
ception
    {
        int size;
        FileIn
```

Eso es todo. Este ejemplo se encuentra en el archivo fileinputstream.java del CD.

Trabajar con FileOutputStream

"Bien", dice el programador novato, "sé que se utiliza FileInputStream para leer desde archivos, pero, ¿qué clase puedo utilizar si quiero escribir en un archivo?" Sonreímos y decimos, "Apuesto a que lo puede adivinar"

Puede utilizar la clase FileOutputStream para escribir datos, byte a byte, a un archivo. Aquí tiene el diagrama de herencia para esa clase, que deriva de la clase OutputStream:

```
java.lang.Object
  |
  +--java.io.OutputStream
        |
        +--java.io.FileOutputStream
```

Aquí tiene los métodos de escritura que puede utilizar con esta clase:

- ***int write(byte[] b)***. Escribe b.length bytes a partir de la matriz de bytes dada a este stream de archivo de salida.

- ***void write(byte[] b, int off, int len)***. Escribe len bytes a partir de la matriz de bytes de entrada, iniciando el desplazamiento en off a este stream de salida de archivo.
- ***void write(int b)***. Escribe el byte dado a este stream de salida de archivo.

Aquí tiene un ejemplo en el que escribimos la cadena de texto "Esto es una cadena de texto." a un archivo de tres formas: byte a byte, todos a la vez y en parte. Aquí tiene la apariencia el código:

```
import java.io.*;

class fileoutputstream
{
    public static void main(String args[]) throws Exception
    {
        byte data[] = "Esto es una cadena de texto.".getBytes();

        FileOutputStream fileoutputstream1 = new
FileOutputStream("file1.txt");
      for (int loop_index = 0; loop_index < data.length; loop_index++) {
            fileoutputstream1.write(data[loop_index]);
        }

        FileOutputStream fileoutputstream2 = new
FileOutputStream("file2.txt");
        fileoutputstream2.write(data);

        FileOutputStream fileoutputstream3 = new
FileOutputStream("file3.txt");
        fileoutputstream3.write(data, 5, 10);

        fileoutputstream1.close();
        fileoutputstream2.close();
        fileoutputstream3.close();
  }
}
```

Eso es todo. Este ejemplo se encuentra en el archivo fileoutputstream.java del CD.

Trabajar con ByteArrayInputStream

"Tengo una matriz de bytes enorme en mi programa", dice el programador novato, "y siempre pierdo la posición en ella. ¡No está bien que no podamos tratar una matriz de bytes como un archivo y abrir un stream de entrada para ella!" "Sí se puede", decimos. El PN se queda asombrado.

Puede utilizar la clase ByteArrayInputStream para abrir un stream de entrada desde una matriz de bytes en memoria. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
  +-java.io.InputStream
        |
        +-java.io.ByteArrayInputStream
```

Aquí tiene un ejemplo. En este caso, únicamente creamos un *buffer* de memoria con el texto "Aquí tenemos una cadena" y después abrimos un ByteArrayInputStream con ese *buffer*, leemos y visualizamos el texto byte a byte:

```
import java.io.*;

class bytearrayinputstream
{
    public static void main(String args[]) throws IOException
    {
        byte data[] = "Aquí tenemos una cadena".getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(data);

        int character;
        while((character = in.read()) != -1){
            System.out.print((char) character);
        }
    }
}
```

Aquí tiene la salida de este código:

```
c:\>java bytearrayniputstream
Aquí tenemos una cadena
```

Eso es todo. Este ejemplo se encuentra en el archivo bytearrayinputstream.java del CD.

Trabajar con ByteArrayOutputStream

"Por tanto, podemos leer bytes con ByteArrayInputStream", dice el programador novato. "¿Podemos escribirlos también?" "Claro", decimos, "con ByteArrayOutputStream".

La clase ByteArrayOutputStream permite escribir stream de bytes a *buffer*. Aquí tiene el diagrama de herencia para esta clase:

```
java.lang.Object
  |
```

```
+—java.io.OutputStream
      |
      +—java.io.ByteArrayOutputStream
```

Aquí tiene un ejemplo. En este caso, estamos escribiendo bytes a ByteArrayOutputStream, escribiendo bytes a un *buffer* de memoria y enviando los bytes a un stream de salida de archivo:

```
import java.io.*;

class bytearrayoutputstream
{
    public static void main(String args[]) throws IOException
    {
        ByteArrayOutputStream bytearrayoutputstream = new
ByteArrayOutputStream();
        byte data[] = "Aquí tenemos una cadena".getBytes();

        bytearrayoutputstream.write(data);

        System.out.println(bytearrayoutputstream.toString());

        byte buffer[] = bytearrayoutputstream.toByteArray();
       for (int loop_index = 0; loop_index < data.length; loop_index++) {
            System.out.print((char) buffer[loop_index]);
        }

      OutputStream fileoutputstream = new FileOutputStream("file.txt");
        bytearrayoutputstream.writeTo(fileoutputstream);
        fileoutputstream.close();
    }
}
```

Este ejemplo se encuentra en el archivo bytearrayoutputstream.java del CD. Aquí tiene la salida de este código:

```
java bytearrayoutputstream
Aquí tenemos una cadena
Aquí tenemos una cadena
```

21 Programación multihilo y animación

En este capítulo, vamos a examinar la programación multihilo de Java y cómo permite la animación de gráficos. Un hilo es un flujo de ejecución de código y, mediante el uso de hilos, podemos hacer que nuestros programas aparentemente realicen varias tareas al mismo tiempo. Por ejemplo, su código podría interaccionar con el usuario mientras realiza tareas en segundo plano de gran consumo de tiempo. Los hilos separados realmente no se ejecutan al mismo tiempo, por supuesto (a menos que tenga una máquina multiprocesadora); en realidad, cada hilo obtiene secuencias de tiempo del mismo procesador. El resultado aparente, no obstante, es que los diversos hilos se ejecutan al mismo tiempo y puede resultar impresionante. De hecho, Java es uno de los pocos lenguajes de programación que soporta multihilos explícitamente.



Truco: Muchos componentes en Java, sobre todo la mayoría de los componentes de la Swing, no son de hilo seguro; es decir, no es seguro utilizarlos de una forma multihilo.

Usar hilos en Java

Cuando se inicia un programa Java, éste tiene un hilo: el hilo principal. Podemos interaccionar con el hilo principal de diversas formas, como vere-

mos en este capítulo, por ejemplo obteniendo o asignando su nombre, deteniéndolo y mucho más. Sin embargo, podemos también iniciar otros hilos. Podemos iniciar el código en esos hilos llamando al método start del objeto y situando el código que queremos que utilice el hilo en el método run.

Existen dos formas para crear nuevos hilos. Una es declarar una clase que extienda Thread. Esta subclase sobrescribiría el método run de la clase Thread y a continuación podemos alojar e iniciar una instancia de esa clase. Por ejemplo, aquí tiene la forma de definir una nueva clase Thread que se inicia cuando crea una instancia:

```
class CustomThread extends Thread
{
    CustomThread()
    {
        // Constructor de inicio estándar
        start();
    }

    public void run()
    {
   // Realice el trabajo para el que se creó este hilo
    }
}
```

También puede crear e iniciar un hilo de este tipo como sigue:

```
CustomThread thread1 = new CustomThread();
```

La otra forma de crear un hilo es declarar una clase que implemente la interfaz Runnable. Esa clase entonces implementará el método run. Una instancia de la clase puede alojarse (es decir, pasarse como argumento cuando crea un objeto Thread). Aquí tiene su apariencia en el código:

```
class CustomThread extends Thread
{
  Thread thread;

    CustomThread()
    {
        // Constructor de inicio estándar
        thread = new CustomThread( this, "segundo");
        thread.start();
    }

    public void run()
    {
   // Realice el trabajo para el que se creó este hilo
    }
}
```

Este código entonces crearía un hilo de este tipo e iniciaría su ejecución:

```
CustomThread thread1 = new CustomThread();
```

También es interesante observar que cada hilo tiene un nombre para propósitos de identificación (de hecho, más de un hilo puede tener el mismo nombre). Cualquier hilo también tiene una prioridad y los hilos con prioridades mayores se ejecutan de forma preferente con respecto a los hilos de prioridad menor.

Esto es todo para el resumen de lo que aparece en este capítulo. Vamos a hacer funcionar la programación multihilo.

Obtener el hilo principal

El programador novato aparece y dice, "¿Hilos? He estado programando Java durante un tiempo y no he visto ningún hilo". Sonreímos y contestamos, "Todo programa Java al menos tiene un hilo, llamado *hilo principal* y que podemos observar con el método currentThread".

El método currentThread de la clase Thread obtiene el hilo actual; si está ejecutando el hilo principal, currentThread devuelve el hilo principal. Aquí tiene un ejemplo en el que se obtiene el hilo principal de una aplicación y se imprime su nombre utilizando el método getName (observe que puede asignar el nombre de un hilo con setName; consulte el siguiente tema):

```
class mainthread
{
    public static void main(String args[])
    {
        Thread thread = Thread.currentThread();

    System.out.println("El hilo principal se llama " + thread.getName());
    }
}
```

Aquí tiene la salida de este código (mainthread.java en el CD que acompaña a este libro):

```
c:\>java mainthread
El hilo principal se llama main
```

Dar nombre a un hilo

Una forma de distinguir los hilos es por su nombre y podemos asignar el nombre de un hilo con el método setName. Aquí tiene un ejemplo en el que

cambiamos el nombre del hilo principal en una aplicación y visualizamos ese nombre:

```
class setname
{
    public static void main(String args[])
    {
        Thread thread = Thread.currentThread();

      System.out.println("El nombre original del hilo principal es " +
            thread.getName());

        thread.setName("El hilo principal");

        System.out.println("El nombre del hilo principal es ahora " +
            thread.getName());

    }
}
```

Aquí tiene el código de salida (setname.java en el CD):

```
c:\>java setname
El nombre original del hilo principal es main
El nombre del hilo principal es ahora El hilo principal
```

Una vez ha dado un nombre a un hilo, puede utilizar el método getName para comprobar el nombre y así determinar cuál es el hilo que está trabajando con una sección de código que ejecutan muchos hilos (como el código del método run de la clase Thread).

Detener un hilo

Algunas veces, deseará detener la ejecución de un hilo y puede hacerlo con el método sleep. Puede pasar a este método la cantidad de tiempo de espera, en milisegundos (un milisegundo es una *milésima de segundo*) y el hilo quedará detenido esa cantidad de tiempo antes de continuar. Aquí tiene un ejemplo en el que imprimimos un mensaje, y detenemos el hilo durante un segundo entre las palabras:

```
class sleep
{
    public static void main(String args[])
    {
        try {
            System.out.println("Hola");
            Thread.sleep(1000);
            System.out.println("desde");
```

```
            Thread.sleep(1000);
            System.out.println("Java.");
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}
```

Aquí tiene el resultado. Cuando ejecute este código (sleep.java en el CD), verá las palabras apareciendo con un intervalo de un segundo entre ellas:

```
c:\>java sleep
Hola
desde
Java.
```

Crear un hilo con la interfaz Runnable

"Quiero crear un nuevo hilo para realizar el análisis sintáctico de un documento mientras el usuario está haciendo otras cosas", dice el programador novato. "¿Cómo lo hago?" "Bien", decimos, "existen dos formas de crear sus propios hilos; implementar la interfaz Runnable y extender la clase Thread". "Mmh", dice el PN, "vamos a comenzar con Runnable".

La sobrescritura de la interfaz Runnable es una de las dos formas de crear sus propios hilos en Java. Esta interfaz únicamente tiene un método: run. Basta situar el código que queremos que ejecute el nuevo hilo en el método anterior. Cuando creemos un nuevo objeto Runnable, podemos pasar ese objeto al constructor de la clase Thread para crear el nuevo hilo.

Aquí tiene un ejemplo en el cual creamos una nueva clase que implementa la interfaz Runnable. Se pasa a sí misma al constructor de la clase Thread y se inicia cuando se crea un objeto de esta clase. Cuando se ejecuta, imprime su nombre una vez por segundo durante diez segundos y después sale. Aquí tiene el código para la clase Runnable, que llamaremos SecondThread (observe que el trabajo actual se realiza en el método run y que el código se ejecuta con el método start de la clase Thread):

```
class SecondThread implements Runnable
{
    Thread thread;

    SecondThread()
    {
        thread = new Thread(this, "segundo");
        System.out.println("Iniciando segundo hilo");
        thread.start();
    }
```

```
    public void run()
    {
        try {
            for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}

        System.out.println("Final del segundo hilo.");
    }
}
```

Aquí tiene la aplicación que creará un nuevo hilo de la clase SecondThread y lo ejecutará; observe también que hacemos que el proceso principal imprima su nombre para que pueda ver la alternancia entre los dos hilos:

```
class runnable
{
    public static void main(String args[])
    {
        SecondThread secondthread = new SecondThread();

        try {
            for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}
    }
}
```

Aquí tiene el resultado de este código (runnable.java en el CD). Observe que los dos hilos alternan la impresión de sus nombres (además, observe que no puede garantizar que quedarán alternados; eso depende del sistema operativo):

```
c:\>java runnable
Iniciando segundo hilo
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
```

```
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
Final del segundo hilo.
```

Crear un hilo con la clase Thread

"Bien", dice el programador novato, "Sé que puedo crear un nuevo hilo utilizando la interfaz Runnable..." "O bien", decimos, "utilizando la clase Thread". "¡Explíqueme más!", dice el PN.

La clase Thread soporta nuevos hilos. También implementa la interfaz Runnable, por lo que no tendremos que hacerlo. Aquí tiene el diagrama de herencia para Thread:

```
java.lang.Object
  |
  +--java.lang.Thread
```

La tabla 21.1 muestra los campos para la clase Thread, la tabla 21.2 sus constructores y la tabla 21.3 sus métodos.

Tabla 21.1. Campos de la clase Thread.

Campo	Descripción
static int MAX_PRIORITY	La prioridad máxima que puede tener un hilo.
static int MIN_PRIORITY	La prioridad mínima que puede tener un hilo.
static int NORM_PRIORITY	La prioridad predeterminada asignada a un hilo.

Tabla 21.2. Constructores de la clase Thread.

Constructor	Descripción
Thread()	Construye un nuevo objeto Thread.
Thread(Runnable target)	Construye un nuevo objeto Thread utilizando un objeto Runnable.

Constructor	Descripción
Thread(Runnable target, String name)	Construye un nuevo objeto Thread con un objeto Runnable y un nombre.
Thread(String name)	Construye un nuevo objeto Thread con un nombre.
Thread(ThreadGroup group, Runnable target)	Construye un nuevo objeto Thread como parte de un grupo.
Thread(ThreadGroup group, Runnable target, String name)	Construye un nuevo objeto Thread con un objeto Runnable. Este objeto Thread tiene el nombre especificado y pertenece al grupo de hilos al que hace referencia group.
Thread(ThreadGroup group, String name)	Construye un objeto Thread que es parte de un grupo y tiene nombre.

Tabla 21.3. Métodos de la clase Thread.

Método	Descripción
static int activeCount()	Obtiene el número actual de hilos activos en el hilo del grupo de hilos.
void checkAccess()	Determina si el hilo actualmente en ejecución tiene permiso para modificar este hilo.
int countStackFrames()	Obsoleto. La definición de esta llamada depende de suspend(), que está obsoleta.
static Thread currentThread()	Obtiene una referencia al objeto Thread actualmente en ejecución.
void destroy()	Destruye este hilo.
static void dumpStack()	Imprime una traza de la pila del hilo actual.
static int enumerate(Thread[] tarray)	Copia a la matriz indicada todos los hilos activos en el hilo del grupo de hilos.
ClassLoader getContextClassLoader()	Obtiene el contexto ClassLoader para este hilo.
String getName()	Obtiene el nombre del hilo.

Método	Descripción
int getPriority()	Obtiene la prioridad del hilo.
ThreadGroup getThreadGroup()	Obtiene el grupo de hilos al que pertenece este hilo.
void interrupt()	Interrumpe este hilo.
static boolean interrupted()	Devuelve true si el hilo actual ha sido interrumpido.
boolean isAlive()	Devuelve true si el hilo está vivo.
boolean isDaemon()	Devuelve true si este hilo es un hilo demonio.
boolean isInterrupted()	Devuelve true si éste hilo ha sido interrumpido.
void join()	Espera hasta que muera este hilo.
void join(long millis)	Espera al menos *millis* milisegundos para que este hilo muera.
void join(long millis, int nanos)	Espera al menos *millis* milisegundos más *nanos* nanosegundos para que este hilo muera.
void resume()	Obsoleto. Este método existe únicamente para utilizar con suspend, que ha quedado obsoleto.
void run()	Si este hilo fue construido con una interfaz Runnable de un objeto run separado, se llama al método run del objeto Runnable.
void setContextClassLoader(ClassLoader cl)	Asigna el contexto ClassLoader para este hilo.
void setDaemon(boolean on)	Marca este hilo como hilo demonio o hilo de usuario.
void setName(String name)	Cambia el nombre de este hilo para que sea igual al nombre del argumento.
void setPriority(int newPriority)	Cambia la prioridad de este hilo.
static void sleep(long millis)	Hace que el hilo en ejecución actualmente duerma durante el número indicado de milisegundos.

Método	Descripción
static void sleep(long millis, int nanos)	Hace que el hilo en ejecución actualmente duerma durante el número indicado de milisegundos más el número indicado de nanosegundos.
void start()	Hace que este hilo comience su ejecución, lo que hace que la máquina virtual Java llame al método run de este hilo.
void stop()	Obsoleto. Este método es considerado inherentemente inseguro.
void stop(Throwable obj)	Obsoleto. Este método es considerado inherentemente inseguro.
void suspend()	Obsoleto este método ha quedado obsoleto debido a que se considera propenso a bloqueos inherentemente.
String toString()	Obtiene una representación de cadena de este hilo.
static void yield()	Hace que el objeto Thread en ejecución actualmente se detenga temporalmente, lo que puede permitir que otros hilos se ejecuten.

Aquí tiene un ejemplo en el que creamos un segundo hilo y hacemos que imprima su nombre diez veces, una por segundo. Observe que, en este caso, estamos llamando al constructor del hilo y pasándole el nombre del segundo hilo para, después, llamar al método start del hilo para iniciar la ejecución del código en el método run. La acción tiene lugar en el método run; ahí es donde el hilo imprime su nombre cada segundo. Aquí tiene el código:

```
class SecondThread extends Thread
{
    SecondThread()
    {
        super("segundo");
        start();
    }

    public void run()
    {
        try {
```

```
            for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}

        System.out.println("Segundo hilo finalizando.");
    }
}
```

Aquí tiene una aplicación en la que utilizamos la clase Thread y también imprime el nombre del hilo principal cada segundo:

```
class thread
{
    public static void main(String args[])
    {
        SecondThread secondthread = new SecondThread();

        try {
            for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}
    }
}
```

Aquí tiene el resultado de este código (thread.java en el CD). Observe que ambos hilos están funcionando y visualizan sus nombres, una vez por segundo:

```
c:\>java thread
Iniciando segundo hilo
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
main hilo aquí...
```

```
segundo hilo aquí...
main hilo aquí...
segundo hilo aquí...
Final del segundo hilo.
```

Crear hilos múltiples

"Sé que puedo iniciar un segundo hilo", dice el programador novato, "pero, ¿qué tal si inicio *cuatro* nuevos hilos?". Sonreímos y contestamos, "No hay problema".

Puede crear y ejecutar múltiples hilos en un programa; basta con dar a cada hilo un nuevo objeto. Aquí tiene un ejemplo en el que creamos cuatro hilos y hacemos que cada uno imprima su nombre una vez por segundo:

```
class CustomThread extends Thread
{
    CustomThread(String name)
    {
        super(name);
        start();
    }

    public void run()
    {
        try {
            for(int loop_index = 0; loop_index < 4; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}

       System.out.println((Thread.currentThread()).getName() + " fina-
liza.");
    }
}

class multithread
{
    public static void main(String args[])
    {
        CustomThread thread1 = new CustomThread("primero");
        CustomThread thread2 = new CustomThread("segundo");
        CustomThread thread3 = new CustomThread("tercero");
        CustomThread thread4 = new CustomThread("cuarto");

        try {
            for(int loop_index = 0; loop_index < 10; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
```

```
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}
    }
}
```

Aquí tiene el resultado de este código (multithread.java en el CD):

```
primero hilo aquí...
main hilo aquí...
segundo hilo aquí...
cuarto hilo aquí...
tercero hilo aquí...
primero hilo aquí...
main hilo aquí...
segundo hilo aquí...
cuarto hilo aquí...
tercero hilo aquí...
primero hilo aquí...
main hilo aquí...
segundo hilo aquí...
cuarto hilo aquí...
tercero hilo aquí...
primero hilo aquí...
main hilo aquí...
segundo hilo aquí...
cuarto hilo aquí...
...
```

Espera (para unión) de hilos

"Necesito más control sobre los hilos", dice el programador novato. "Por ejemplo, necesito ser capaz de esperar hasta que un hilo termine la ejecución antes de seguir con el resto del programa. ¿Existe alguna forma de comprobarlo?". "Claro", decimos, "puede utilizar el método join para esperar hasta que un hilo termine". "¿De verdad? ", pregunta interesado el PN.

El método join de la clase Thread espera hasta que se termine la ejecución de un hilo (también conocido como *espera hasta la muerte de un hilo*) antes de retornar. Aquí tiene un ejemplo en el que creamos cuatro nuevos hilos y esperamos a que termine cada uno antes de finalizar la aplicación principal:

```
class CustomThread extends Thread
{
    CustomThread(String name)
    {
        super(name);
        start();
    }

    public void run()
```

```
        {
            try {
                for(int loop_index = 0; loop_index < 4; loop_index++) {
                    System.out.println((Thread.currentThread()).getName()
                        + " hilo aquí...");
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {}

          System.out.println((Thread.currentThread()).getName() + " fina-
liza.");
        }
    }

    class join
    {
        public static void main(String args[])
        {
            CustomThread thread1 = new CustomThread("primer");
            CustomThread thread2 = new CustomThread("segundo");
            CustomThread thread3 = new CustomThread("tercero");
            CustomThread thread4 = new CustomThread("cuarto");

            try {
                thread1.join();
                thread2.join();
                thread3.join();
                thread3.join();
            } catch (InterruptedException e) {}
        }
    }
```

Aquí tiene la salida de este código (join.java en el CD):

```
c:\>java join
primer hilo aquí...
segundo hilo aquí...
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
segundo hilo aquí...
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
segundo hilo aquí...
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
tercero hilo aquí...
segundo hilo aquí...
cuarto hilo aquí...
primer finaliza.
tercero finaliza.
segundo finaliza.
cuarto finaliza.
```

Comprobar que un hilo está vivo

"Oh, oh", dice el programador novato, "he perdido el hilo. ¿Cómo sé si todavía está realizando algo?. "Ningún problema en absoluto", decimos. "Para comprobar un hilo, puede utilizar su método isAlive para ver si el hilo todavía está activo". "¿Cómo funciona?", pregunta el PN.

Aquí tiene un ejemplo en el que creamos y ejecutamos cuatro nuevos hilos. A medida que los hilos se ejecutan, utilizamos el método isAlive del primer hilo nuevo y, cuando todos los hilos han terminado (lo que verificamos con el método join), comprobamos de nuevo ese proceso con isAlive. Aquí tiene el código:

```
class CustomThread extends Thread
{
    CustomThread(String name)
    {
        super(name);
        start();
    }

    public void run()
    {
        try {
            for(int loop_index = 0; loop_index < 4; loop_index++) {
                System.out.println((Thread.currentThread()).getName()
                    + " hilo aquí...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}

       System.out.println((Thread.currentThread()).getName() + " fina-
liza.");
    }
}

class isalive
{
    public static void main(String args[])
    {
        CustomThread thread1 = new CustomThread("primer");
        CustomThread thread2 = new CustomThread("segundo");
        CustomThread thread3 = new CustomThread("tercero");
        CustomThread thread4 = new CustomThread("cuarto");

        System.out.println(thread1.isAlive());

        try {
            thread1.join();
            thread2.join();
            thread3.join();
```

```
            thread3.join();
        } catch (InterruptedException e) {}

        System.out.println(thread1.isAlive());
    }
}
```

Aquí tiene la salida de este código (isalive.java en el CD). Observe que el valor que devuelve isAlive, es true cuando thread1 está en ejecución y false después de que el proceso haya salido:

```
c:\>java isalive
true
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
segundo hilo aquí...
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
segundo hilo aquí...
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
segundo hilo aquí...
tercero hilo aquí...
cuarto hilo aquí...
primer hilo aquí...
segundo hilo aquí...
tercero finaliza.
cuarto finaliza.
primer finaliza.
segundo finaliza.
false
```

Asignar la prioridad y detención de hilos

"Mmh", dice el programador novato, "sé cómo suspender hilos, pero hay un problema. Estoy trabajando en el nuevo programa *SuperDuperMultiThreadedDataCrunch* y algunas tareas son más importantes que otras. Por ejemplo, quiero descargar imágenes que el usuario ha solicitado antes de realizar una comprobación gramatical en segundo plano". "Bien", decimos, "puede dar a los hilos distintas prioridades". El PN dice: "¡Explíquemelo!".

Cada hilo en Java tiene una prioridad, que está entre Thread.MAX_PRIORITY y Thread.MIN_PRIORITY. Aquí tiene las prioridades que están definidas actualmente:

- MAX_PRIORITY. La prioridad máxima que puede tener un hilo.
- MIN_PRIORITY. La prioridad mínima que puede tener un hilo.
- NORM_PRIORITY. La prioridad predeterminada asignada a un hilo.

El valor de Thread.MAX_PRIORITY es 10, Thread.MIN_PRIORITY es uno y NORM_PRIORITY es 5.

La asignación de prioridades es una tarea muy complicada debido a que depende del sistema operativo y los resultados reales dependen en gran manera de otras aplicaciones que se ejecuten al mismo tiempo. De todas formas, puede asignar las prioridades relativas de los hilos. Aquí tiene un ejemplo en el que proporcionaremos cuatro diferentes prioridades a cuatro hilos y les haremos que cuenten durante cinco segundos. Entonces los finalizaremos y visualizaremos su resultado para obtener una indicación de la velocidad de ejecución relativa. Aquí tiene el código (observe que para los hilos, utilizamos un semáforo que se comprueba en el método run y queda definido como *volatile*, lo que significa que su valor puede ser cambiado por cualquier otro hilo):

```
class Counter implements Runnable
{
    Thread thread;
    int counter = 0;
    volatile boolean goflag;

    public Counter(int p)
    {
        thread = new Thread(this);
        thread.setPriority(p);
    }

    public void start()
    {
        goflag = true;
        thread.start();
    }

    public void run()
    {
        while (goflag) counter++;
    }

    public void end() {
        goflag = false;
    }

}

class priority
```

```
{
    public static void main(String args[])
    {
        Counter thread1 = new Counter(Thread.NORM_PRIORITY + 2);
        Counter thread2 = new Counter(Thread.NORM_PRIORITY + 1);
        Counter thread3 = new Counter(Thread.NORM_PRIORITY - 1);
        Counter thread4 = new Counter(Thread.NORM_PRIORITY - 2);

        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {}

        thread1.end();
        thread2.end();
        thread3.end();
        thread4.end();

        System.out.println("Hilo 1 contado: " + thread1.counter);
        System.out.println("Hilo 2 contado: " + thread2.counter);
        System.out.println("Hilo 3 contado: " + thread3.counter);
        System.out.println("Hilo 4 contado: " + thread4.counter);
    }
}
```

Para detener los hilos, llamamos al método end personalizado, que asigna a un indicador llamado goFlag el valor false. El cual, a su vez, detiene el código en el método run.

En los primeros días de Java, se podía detener un hilo con el método integrado stop, pero ese método ha quedado obsoleto ahora, debido a que no es seguro; puede dejar que el sistema operativo acceda a los monitores en un estado no estable.

En su lugar, para detener un hilo, puede utilizar ahora el método precedente, es decir, comprobar algún indicador que pueda asignar desde fuera del hilo.

Aquí tiene la salida de este código (priority.java en el CD). Observe el distinto número de cuentas para los hilos con distintas prioridades (sus resultados pueden variar, por supuesto, dependiendo de su sistema operativo y la velocidad de la máquina):

```
c:\>java priority
Hilo 1 contado: 258865319
Hilo 2 contado: 239589379
Hilo 3 contado: 27347113
Hilo 4 contado: 25558903
```

¿Por qué utilizar la sincronización?

"Bien", dice el programador novato, "creo que no quiero usar más hilos. Tengo un objeto que almacena los datos de mi programa y no puedo tener muchos hilos distintos trabajando con el mismo objeto a la vez; ¿qué sucede si un hilo cambia parte del objeto de datos y después otro hilo cambia la misma parte del objeto de datos antes de que el primer hilo haya terminado? ¡El caos!" "Bien", sonreímos, "puede impedir que los nuevos hilos trabajen con objetos como ese hasta que el hilo actual haya terminado, sincronizando sus hilos". "¡Vaya!", dice el PN.

Excluir otros hilos de una operación hasta que la operación se haya completado se conoce como *sincronización*. Aquí tiene un ejemplo que muestra por qué resulta útil la sincronización. En este caso, tenemos cuatro nuevos hilos que trabajan todos con un objeto de datos compartido. Este objeto imprime un mensaje de inicio y un mensaje de fin y completa una tarea de medio segundo entre los mensajes. Realmente, el objeto de datos debería iniciarse, realizar su tarea y terminar antes de que el siguiente hilo inicie una nueva tarea, pero ésa no es la forma en que trabajamos aquí; cada hilo bloquea el objeto de datos para sí mismo. Aquí tiene el código:

```
class synchronize1
{
    public static void main(String args[])
    {
        Shared shared = new Shared();

        CustomThread thread1 = new CustomThread(shared, "uno");
        CustomThread thread2 = new CustomThread(shared, "dos");
        CustomThread thread3 = new CustomThread(shared, "tres");
        CustomThread thread4 = new CustomThread(shared, "cuatro");

        try {
            thread1.join();
            thread2.join();
            thread3.join();
            thread4.join();
        } catch(InterruptedException e) {}
    }
}

class CustomThread extends Thread
{
    Shared shared;

    public CustomThread(Shared shared, String string)
    {
        super(string);
```

```
        this.shared = shared;
        start();
    }

    public void run() {
        shared.doWork(Thread.currentThread().getName());
    }
}

class Shared
{
    void doWork(String string)
    {
        System.out.println("Iniciando " + string);

        try {
            Thread.sleep((long) (Math.random() * 500));
        } catch (InterruptedException e) {}

        System.out.println("Terminando " + string);
    }
}
```

Aquí tiene la salida de este código (synchronize1.java en el CD). Observe que cada tarea se superpone con las otras:

```
c:\>java synchronize1
Iniciando dos
Iniciando tres
Iniciando cuatro
Iniciando uno
Terminando cuatro
Terminando dos
Terminando uno
Terminando tres
```

He aquí un problema si está trabajando con los datos reales: ¿qué sucede si lee algún dato del objeto de datos, o trabaja con él y lo rescribe, pero otro hilo ya ha realizado la misma cosa? Cuando escribimos los datos, los cambios realizados por otro hilo se perderán. Como puede ver, es importante ser capaz de bloquear el acceso a otros hilos para los recursos críticos en un momento dado. Examine los siguientes dos temas para ver cómo funciona esto.

Sincronizar bloques de código

"Bien", dice el programador novato, "sé que puede existir un problema cuando múltiples hilos acceden al mismo objeto o recurso, pero, ¿cómo bloqueo el acceso a nuevos hilos hasta que el actual haya terminado con ese

recurso u objeto?" "Existen dos formas", decimos. "Bien", dice el PN, "¿cuál es la primera?".

Existen dos formas de sincronizar la ejecución del código de hilos: la sincronización de bloques de código y los métodos sincronizados. Examinaremos los bloques sincronizados de código en este tema y los métodos sincronizados en el siguiente. Para sincronizar un bloque de código, utilizamos la palabra reservada *synchronize*, indicando el objeto al que queremos restringir el acceso. Si extendemos el ejemplo del tema previo, el proceso tiene la siguiente apariencia:

```
class synchronize2
{
    public static void main(String args[])
    {
        Shared shared = new Shared();

        CustomThread thread1 = new CustomThread(shared, "uno");
        CustomThread thread2 = new CustomThread(shared, "dos");
        CustomThread thread3 = new CustomThread(shared, "tres");
        CustomThread thread4 = new CustomThread(shared, "cuatro");

        try {
            thread1.join();
            thread2.join();
            thread3.join();
            thread4.join();
        } catch(InterruptedException e) {}
    }
}

class CustomThread extends Thread
{
    Shared shared;

    public CustomThread(Shared shared, String string)
    {
        super(string);
        this.shared = shared;
        start();
    }

    public void run() {
        synchronized(shared) {
            shared.doWork(Thread.currentThread().getName());
        }
    }
}

class Shared
{
    void doWork(String string)
    {
```

```
        System.out.println("Iniciando " + string);

        try {
            Thread.sleep((long) (Math.random() * 500));
        } catch (InterruptedException e) {}

        System.out.println("Finalizando " + string);
    }
}
```

Aquí tiene la salida de este código (synchronize2.java en el CD). Observe que cada tarea se inicia y termina antes de que cualquier otro hilo inicie una nueva tarea:

```
c:\>java synchronize2
Iniciando uno
Finalizando uno
Iniciando dos
Finalizando dos
Iniciando tres
Finalizando tres
Iniciando cuatro
Finalizando cuatro
```

Métodos sincronizados

Además de los bloques de código sincronizados (tema anterior), también puede bloquear el acceso a otros hilos sincronizando métodos. En este caso, utilice la palabra reservada *synchronize* cuando defina un método. Aquí tiene la forma de modificar el método doWork en el objeto *shared* del ejemplo del tema anterior al previo para sincronizar los hilos:

```
class synchronize3
{
    public static void main(String args[])
    {
        Shared shared = new Shared();

        CustomThread thread1 = new CustomThread(shared, "uno");
        CustomThread thread2 = new CustomThread(shared, "dos");
        CustomThread thread3 = new CustomThread(shared, "tres");
        CustomThread thread4 = new CustomThread(shared, "cuatro");

        try {
            thread1.join();
            thread2.join();
            thread3.join();
            thread4.join();
        } catch(InterruptedException e) {}
    }
```

```
}

class CustomThread extends Thread
{
    Shared shared;

    public CustomThread(Shared shared, String string)
    {
        super(string);
        this.shared = shared;
        start();
    }

    public void run() {
            shared.doWork(Thread.currentThread().getName());
    }
}

class Shared
{
    synchronized void doWork(String string)
    {
        System.out.println("Iniciando " + string);

        try {
            Thread.sleep((long) (Math.random() * 500));
        } catch (InterruptedException e) {}

        System.out.println("Finalizando " + string);
    }
}
```

Ahora, únicamente un hilo cada vez puede entrar en el método doWork. Aquí tiene la salida de este código (synchronize3.java en el CD):

```
c:\>java synchronize3
Iniciando uno
Finalizando uno
Iniciando dos
Finalizando dos
Iniciando tres
Finalizando tres
Iniciando cuatro
Finalizando cuatro
```

Comunicación entre hilos

"Maldición", dice el programador novato, "tengo un problema con los hilos. Parte de mi programa está escribiendo datos y parte leyéndolos; pero ¡unas veces la parte de lectura se adelanta a la parte de escritura!". "Ese es un

problema clásico de producción/consumición", decimos, "y puede arreglarlo con los métodos wait y notify". El PN sonríe y dice, "¿Sí?".

A veces, los hilos necesitarán combinarse entre ellos, especialmente cuando la salida de un hilo se utiliza en otro hilo. Una forma de coordinar estos hilos es utilizar los métodos notify, wait y notifyAll:

- ***wait***. Hace que un hilo duerma hasta que se llame a notify o notifyAll en el mismo objeto.
- ***notify***. Inicia el primer hilo que llamó a wait en el mismo objeto.
- ***notifyAll***. Inicia todos los hilos que llamaron a wait en el mismo objeto.

El procedimiento habitual para el hilo de lectura será llamar a wait, y para el hilo de escritura llamar a notify, cuando los datos que desea leer el lector estén listos. Aquí tiene un ejemplo. En este caso, un hilo de escritura llamará al método doWork del objeto, que es un método que consume tiempo, escribe algún dato, y un hilo lector llamará al mismo método getResult del objeto para leer los resultados. Queremos que el hilo lector tenga que esperar hasta que doWork haya terminado. Por tanto, todo lo que tenemos que hacer es llamar a wait en getResult para hacer que el lector espere y llame a notify en doWork cuando el hilo que escribe termine y los datos estén listos para la lectura. Aquí tiene la apariencia del código:

```
class Shared
{
    int data = 0;

    synchronized void doWork()
    {
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {}

        data = 1;
        notify();
    }

    synchronized int getResult() {
        try {
            wait();
        } catch(InterruptedException e) {}

        return data;
    }
}

class CustomThread1 extends Thread
{
```

```
    Shared shared;

    public CustomThread1 (Shared shared, String string)
    {
        super(string);
        this.shared = shared;
        start();
    }

    public void run() {
        System.out.println("El resultado es " + shared.getResult());
    }
}

class CustomThread2 extends Thread
{
    Shared shared;

    public CustomThread2 (Shared shared, String string)
    {
        super(string);
        this.shared = shared;
        start();
    }

    public void run() {
        shared.doWork();
    }
}


class wait
{
    public static void main(String args[])
    {
        Shared shared = new Shared();
        CustomThread1 thread1 = new CustomThread1(shared, "uno");
        CustomThread2 thread2 = new CustomThread2(shared, "dos");
    }
}
```

Aquí tiene la salida de este código (wait.java en el CD). Observe que el hilo de lectura espera realmente hasta que el hilo que escribe haya terminado:

```
c:\>java wait
El resultado es 1
```

Suspender y reanudar hilos

Anteriormente, los hilos en Java soportaban tanto el método *suspend* como el *resume*, que se podían utilizar para detener e iniciar de nuevo tempo-

ralmente un proceso. Desgraciadamente, como sucede con el método *stop*, tanto *suspend* como *resume* han sido descatalogados. En este caso, estos métodos se han descatalogado debido a que eran propensos al bloqueo y podían bloquear el acceso a otros hilos. No obstante, puede crear su propio método *suspend* y *resume* y mostraremos cómo hacerlo aquí.

Observe que cuando subclasifica la clase Thread, no puede llamar a sus nuevos métodos de reanudación *suspend* y *resume*, debido a que Java se quejará de que intente sobrescribir métodos descatalogados. Por tanto, en este ejemplo, llamaremos a estos métodos *newSuspend* y *newResume* (esto no es un problema cuando se trabaja con un objeto que implementa, en cambio, la clase Runnable). Como es el caso en que implementa un nuevo método *stop* (consulte el tema correspondiente en este mismo capítulo), estos métodos trabajan asignando un indicador que se comprueba en el método run. Cuando asignamos ese indicador al valor true en *newSuspend*, utilizamos el método *wait* del método run para suspender el hilo y reanudar el hilo de nuevo con el método *notify* en el método *newResume*. Aquí tiene la apariencia del código:

```
class CustomThread extends Thread
{
    volatile boolean goFlag = true;

    CustomThread(String name)
    {
        super(name);
        start();
    }

    public void run()
    {
        try {
            for(int loop_index = 0; loop_index <= 5; loop_index++) {
                System.out.println(Thread.currentThread().getName() +
" aquí...");
                Thread.sleep(500);
                synchronized(this) {
                    while(!goFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {}
    }

    public void newSuspend()
    {
        goFlag = false;
    }

    synchronized public void newResume()
```

```
    {
        goFlag = true;
        notify();
    }
}

class suspend
{
    public static void main(String args[])
    {
        CustomThread thread1 = new CustomThread("uno");
        CustomThread thread2 = new CustomThread("dos");

        try {
            Thread.sleep(1000);
            System.out.println("Suspendiendo hilo uno...");
            thread1.newSuspend();
            Thread.sleep(1000);
            System.out.println("Resumiendo hilo uno...");
            thread1.newResume();
        } catch (InterruptedException e) {}

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {}
    }
}
```

Aquí tiene la salida del código (suspend.java en el CD). Observe que los hilos uno y dos se ejecutan hasta que el código suspende el hilo uno; entonces cuando el código inicia de nuevo el hilo uno, alternan otra vez:

```
c:\>java suspend
uno aquí...
dos aquí...
uno aquí...
dos aquí...
Suspendiendo hilo uno...
dos aquí...
dos aquí...
Resumiendo hilo uno...
uno aquí...
dos aquí...
uno aquí...
dos aquí...
uno aquí...
uno aquí...
```

Crear gráficos animados con hilos

"Digamos", dice el programador novato, "que he pensado algo, utilizar hilos es ideal cuando quieres dar soporte a animación de gráficos, ¿verdad?".

"Así es", decimos. "Fantástico", dice el PN. "Escriba el código, que yo miraré".

Aquí tiene un ejemplo que muestra cómo puede funcionar la animación de gráficos utilizando hilos. En este applet, cargamos cuatro imágenes en una matriz de imágenes y después realizamos un ciclo a través de ellas en el método paint, haciendo que Java llame a ese método repetidas veces llamando a repaint en un nuevo hilo.

Existe un punto interesante aquí; aunque el método stop de la clase Thread ha sido descatalogado, los navegadores y el *appletviewer* de *Sun* todavía llaman a este método cuando la página que contiene el applet ya no es la actual. Para detener el hilo de animación del applet y que no utilice recursos del sistema hasta que el navegador decida descargarlo, puede situar código en el método stop; pero eso significa que no puede basar su código en la clase Thread, debido a que stop está descatalogado allí. Por tanto, escribiremos un ejemplo utilizando la interfaz Runnable, que no tiene un método stop descatalogado. Aquí tiene el código:

```
import java.awt.*;
import java.applet.Applet;

    /*
<APPLET
    CODE = whirl1.class
    WIDTH = 300
    HEIGHT = 300>
</APPLET>
    */

public class whirl1 extends Applet implements Runnable
{
    Image whirlImages[] = new Image[4];
    Image nowImage;
    int whirlIndex = 0;
    Thread whirlThread;
    boolean animateFlag = true;

    public void init()
    {
        whirlImages[0] = getImage(getCodeBase(), "whirl1.gif");
        whirlImages[1] = getImage(getCodeBase(), "whirl2.gif");
        whirlImages[2] = getImage(getCodeBase(), "whirl3.gif");
        whirlImages[3] = getImage(getCodeBase(), "whirl4.gif");
    }

    public void start()
    {
        whirlThread = new Thread(this);
        whirlThread.start();
```

```
    }

    public void stop()
    {
        animateFlag = false;
    }

    public void run()
    {
        while(animateFlag){
        nowImage = whirlImages[whirlIndex++];
        if(whirlIndex > 3)whirlIndex = 0;
        repaint();
        try {Thread.sleep(200);}
        catch(InterruptedException e) { }
        }
    }

    public void paint (Graphics g)
    {
        if(nowImage != null) g.drawImage(nowImage, 10, 10, this);
    }
}
```

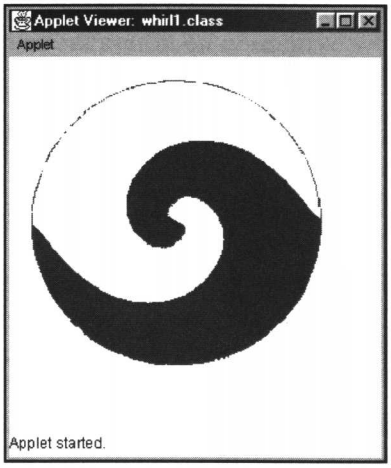


Figura 21.1. Visualización de una animación gráfica.

La figura 21.1 muestra el resultado de este código. Este applet hace que la imagen que se muestra en la figura rote; el código está en el archivo whirl1.java en el CD.

No obstante, existe un problema aquí; he implementado este ejemplo como un applet AWT y cada vez que la imagen se pinta de nuevo, el fondo que existe debajo de la imagen se pinta también, por lo que la animación parpadea mucho. Para arreglar este problema, consulte el siguiente tema.

Eliminar el parpadeo en animaciones gráficas

"Mmh", dice el programador novato, "ahora puedo crear una animación gráfica utilizando hilos, pero seguro que parpadea mucho". "No hay problema", decimos. "Basta sobrescribir el método update". El PN pregunta: "¿Cómo es eso?"

En la programación AWT, cuando se repinta una imagen, su fondo se repinta primero, lo que provoca mucho parpadeo en una animación gráfica. La forma de arreglar esto es sobrescribir el método update, que es el que realiza el repintado. Cuando se sobrescribe update, el fondo de la imagen no se repinta. De hecho, puede hacer más; puede hacer que el método paint repinte únicamente el área cubierta por la imagen que quiere volver a pintar, utilizando el método clipRect de la clase Graphics, como sigue:

```
import java.awt.*;
import java.applet.Applet;

    /*
<APPLET
    CODE = whirl2.class
    WIDTH = 300
    HEIGHT = 300 >
</APPLET>
    */

public class whirl2 extends Applet implements Runnable
{
    Image whirlImages[] = new Image[4];
    Image nowImage;
    int whirlIndex = 0;
    Thread whirlThread;
    boolean animateFlag = true;

    public void init() {
        whirlImages[0] = getImage(getCodeBase(), "whirl1.gif");
        whirlImages[1] = getImage(getCodeBase(), "whirl2.gif");
        whirlImages[2] = getImage(getCodeBase(), "whirl3.gif");
        whirlImages[3] = getImage(getCodeBase(), "whirl4.gif");
    }

    public void start() {
        whirlThread = new Thread(this);
        whirlThread.start();
    }

    public void stop() {
        animateFlag = false;
```

```
    }

    public void run() {
        while(animateFlag){
            nowImage = whirlImages[whirlIndex++];
            if(whirlIndex > 3)whirlIndex = 0;
            repaint();
            try {Thread.sleep(200);}
            catch(InterruptedException e) { }
        }
    }

    public void paint (Graphics g)
    {
        if(nowImage != null) g.drawImage(nowImage, 10, 10, this);
    }

    public void update(Graphics g)
    {
        g.clipRect(10, 10, 280, 280);
        paint(g);
    }
}
```

Ahora la imagen que muestra la figura 21.1 rotará prácticamente libre de parpadeo. Este ejemplo se encuentra en el archivo whirl2.java del CD.

Suspender y reanudar animaciones gráficas

"¿Qué sucede si," dice el programador novato, "deseo suspender y reanudar una animación gráfica? ¿Puedo hacerlo?". "Claro", decimos, "sin problema. Basta utilizar las técnicas estándar de suspensión y reanudación". "Fantástico", dice el PN. "Muéstreme cómo".

Aquí tiene un ejemplo en el cual añadimos botones para suspender y reanudar la animación desarrollada en los dos temas previos. Este ejemplo utiliza las técnicas de suspensión y reanudación tratadas anteriormente en este capítulo (es decir, utiliza un indicador *boolean* y los métodos wait y notify):

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

    /*
<APPLET
    CODE = whirl3.class
```

```
    WIDTH = 300
    HEIGHT = 300 >
</APPLET>
    */

public class whirl3 extends Applet implements ActionListener, Runnable
{
    Image whirlImages[] = new Image[4];
    Image nowImage;
    int whirlIndex = 0;
    Thread whirlThread;
    Button suspendButton, resumeButton;
    boolean animateFlag = true;
    boolean goFlag = true;

    public void init()
    {
        whirlImages[0] = getImage(getCodeBase(), "whirl1.gif");
        whirlImages[1] = getImage(getCodeBase(), "whirl2.gif");
        whirlImages[2] = getImage(getCodeBase(), "whirl3.gif");
        whirlImages[3] = getImage(getCodeBase(), "whirl4.gif");
        suspendButton = new Button("Suspender");
        add(suspendButton);
        suspendButton.addActionListener(this);
        resumeButton = new Button("Resumir");
        add(resumeButton);
        resumeButton.addActionListener(this);
    }

    public synchronized void actionPerformed(ActionEvent e){
         if(e.getSource() == suspendButton){
         goFlag = false;
         }
         if(e.getSource() == resumeButton){
         goFlag = true;
         notify();
         }
    }

    public void start() {
        whirlThread = new Thread(this);
        whirlThread.start();
    }

    public void stop() {
        animateFlag = false;
    }

    public void run() {
        while(animateFlag){
            nowImage = whirlImages[whirlIndex++];
            if(whirlIndex > 3)whirlIndex = 0;
            repaint();
            try {
                Thread.sleep(200);
```

```
                synchronized(this){
                while(!goFlag)
                    wait();
                }
            }
            catch(InterruptedException e) { }
        }
    }

    public void paint (Graphics g)
    {
        if(nowImage != null) g.drawImage(nowImage, 10, 10, this);
    }

    public void update(Graphics g)
    {
        g.clipRect(10, 10, 280, 280);
        paint(g);
    }
}
```

La figura 21.2 muestra el resultado de este código (whirl3.java en el CD).

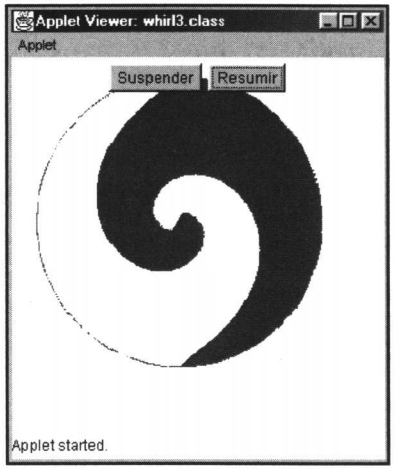


Figura 21.2. Visualización de una animación gráfica con los botones para suspender y reanudar.

Doble buffer

El proceso de *doble buffer* es el que permite la creación de imágenes fuera de pantalla y el volcado a la misma cuando es necesario, lo que implica que no tiene que crear la imagen en el momento en que el usuario la ve. Puede crear un buffer gráfico creando un objeto Image y usando después el método

getGraphics del objeto para obtener un objeto Graphics para la imagen que puede utilizar para dibujar. Aquí tiene un ejemplo que muestra cómo funciona el doble buffer. Este ejemplo dibuja una sucesión de cajas fuera de pantalla y después las visualiza, creando, de esta forma, una animación:

```
import java.awt.*;
import java.applet.Applet;

    /*
  <APPLET
      CODE=dbuffer.class
      WIDTH=200
      HEIGHT=200>
  </APPLET>
    */

public class dbuffer extends Applet implements Runnable
{
    Image image1;
    Thread thread1;
    Graphics graphics;
    int loop_index = 0;
    boolean goFlag = true;

    public void init()
    {
        image1 = createImage(100, 100);
        graphics = image1.getGraphics();
    }

    public void start()
    {
        thread1 = new Thread(this);
        thread1.start();
    }

    public void stop()
    {
        goFlag = false;
    }

    public void run()
    {
        while(goFlag){
            repaint();
            try {Thread.sleep(100);}
            catch(InterruptedException e) {}
        }
    }

    public void paint (Graphics g)
    {
        loop_index += 5;
```

```
            if(loop_index >= 100) loop_index = 5;

            graphics.setColor(new Color(255, 255, 255));
            graphics.fillRect(0, 0, 100, 100);
            graphics.setColor(new Color(0, 0, 0));
            graphics.drawRect(0, 0, loop_index, loop_index);

            g.drawImage(image1, 10, 10, this);
        }
    }
```

22 Creación de paquetes, interfaces, archivos JAR y Java Beans

Crear un paquete

"Vaya", dice el programador novato, "tengo tantas clases que las cosas se han desordenado bastante. ¿Cómo puedo resolver este desorden?" "Bien", decimos, "¿por qué no divide las clases en paquetes? Eso le permitirá distribuir sus archivos de clase en una estructura de directorios de forma similar a cuando distribuye los archivos porque tiene demasiados. Problema resuelto". "Fantástico", dice el PN. "Cuéntaeme cómo funciona".

Cuando tenemos varios archivos de clase, es buena idea distribuirlos en el disco utilizando una jerarquía de directorios. De hecho, los paquetes Java fueron diseñados originalmente para reflejar esa organización de archivos. Puede distribuir archivos de clase en una jerarquía de directorios y permitir que Java sepa lo que sucede con los paquetes.

Por ejemplo, si tiene un paquete llamado package1, que contiene una clase app.class, el archivo de clase iría en un directorio llamado package1, como sigue:

```
package1
-app
```

Múltiples archivos de clase en el mismo paquete irán en el mismo directorio:

```
package1
-app1
-app2
-app3
```

Como el directorio package1 se encuentra en una localización que Java buscará (consulte el capítulo 1 para obtener más información sobre dónde buscará Java los archivos de clase y cómo utilizar la variable de entorno Classpath), Java buscará los archivos de clase que utilice como parte del paquete en ese directorio.

Tiene que indicarle a qué paquete pertenece un archivo de clase, utilizando la sentencia package en su código. Aquí tiene un ejemplo donde creamos app.class e indicamos que forma parte de package1 utilizando la sentencia package:

```
package package1;
public class app
{
  public static void main (String[] args)
  {
      System.out.println ("¡Hola desde Java!");
  }
}
```

Una vez compilada app.class y almacenada en el directorio package1, podemos entonces importar la clase app en el código con sentencias como la siguiente, como haríamos con cualquier otro paquete Java:

```
import package1.*;
import package1.app;
```

La herramienta de tiempo de ejecución de Java también conoce los paquetes; por tanto, debido a que la clase app.class es una aplicación en sí, podemos ejecutar esta clase con una línea de comando que especifique el paquete de la clase app y utilizando un punto (.) para el separador de directorios:

```
c:\> java package1.app
```

De hecho, el separador de paquetes *punto* es útil cuando creamos paquetes dentro de paquetes; examine el siguiente tema para más detalles.

Crear paquetes que contienen paquetes

"Mmh", dice el programador novato, "ya veo que puede poner archivos de clase en un paquete asignando la estructura correcta de directorios y utilizando la sentencia package, pero, ¿qué sucede si tenemos paquetes dentro de paquetes? ¿Qué profundidad puede tener la estructura de anidación de paquetes?" "Tan profunda como quieras", decimos, "es decir, tan profunda como su sistema operativo soporte la anidación de directorios".

Cuando tenemos gran cantidad de archivos de clase, podemos organizarlos en una estructura de paquetes bastante compleja, y lo hacemos creando la estructura de directorios correspondiente en el disco, incluyendo las estructuras de subdirectorios dadas.

Por ejemplo, si quiere que la clase app esté en el paquete package2, que a su vez está dentro del paquete package1, ésta será la estructura de directorios:

```
package1
—package2
——app
```

Para crear esta estructura de paquetes en el código, basta utilizar el separador de paquetes *punto*. Aquí tiene un ejemplo:

```
package package1.package2;

public class app
{
  public static void main (String[] args)
  {
      System.out.println ("¡Hola desde Java!");
  }
}
```

Ahora puede importar la clase app en su código con sentencias como éstas:

```
import package1.package2.*;
import package1.package2.app;
```

Debido a que en este caso app ya es una aplicación clase, también puede ejecutarla como sigue, especificando la estructura de directorios de paquete:

```
c:\> java package1.package2.app
¡Hola desde Java!
```

Crear una interfaz

"Bien", dice el programador novato, "quiero derivar una nueva clase desde dos clases base, pero sé que Java no soporta la herencia múltiple, por lo tanto..." "Por lo tanto tiene que utilizar interfaces", decimos.

Hemos utilizado interfaces a lo largo de este libro y examinaremos cómo crear una ahora. Cuando creamos una interfaz, especificamos los métodos de la interfaz y cuando la implementamos, proporcionamos el código de estos métodos.

Aquí tiene un ejemplo en el que creamos una interfaz llamada Printem que tiene un método: printText. Implementamos esta interfaz en la clase llamada class1. Para crear una interfaz, utilizamos simplemente la sentencia interface y listamos los prototipos (declaraciones sin cuerpo) de los métodos que queremos en la interfaz, como sigue:

```
interface Printem
{
    void printText();
}
```

Ahora podemos implementar esta interfaz en class1:

```
interface Printem
{
    void printText();
}

public class interfaces
{
    public static void main(String[] args)
    {
        class1 object1 = new class1();
        object1.printText();
    }
}

class class1 implements Printem
{
    public void printText()
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Aquí tiene la salida del código (interfaces.java en el CD que acompaña a este libro):

```
c:\> java interfaces
¡Hola desde Java!
```

Implementación parcial de una interfaz

Puede crear clases que implementen parcialmente una interfaz, pero estas clases deben declararse abstractas debido a que son únicamente implementaciones parciales y, por tanto, no pueden instanciarse en objetos.

Aquí tiene un ejemplo en el que creamos una interfaz con dos métodos, printText1 y printText2:

```
interface Printem
{
    void printText1();
    void printText2();
}
```

A continuación, creamos una nueva clase, class1, que implementa la interfaz pero únicamente define un método, printText2 (observe que debido a que esta clase no es una implementación completa de la interfaz, debemos declararla abstracta):

```
interface Printem
{
    void printText1();
    void printText2();
}

abstract class class1 implements Printem
{
    public void printText1()
    {
        System.out.println("¡Hola desde Java!");
    }
}
```

Finalmente, podemos crear una nueva clase, class2, que extienda class1 e implemente printText2, lo que significa que podemos instanciar objetos de la clase class2 como se muestra aquí:

```
interface Printem
{
    void printText1();
    void printText2();
}

public class interfaces2
{
    public static void main(String[] args)
    {
        class2 object2 = new class2();
        object2.printText2();
    }
```

```
}

abstract class class1 implements Printem
{
    public void printText1()
    {
        System.out.println("¡Hola desde Java!");
    }
}

class class2 extends class1
{
    public void printText2()
    {
        System.out.println("¡Hola desde los interfaces Java!");
    }
}
```

Aquí tiene la salida de este código (interfaces2.java en el CD):

```
c:\> java interfaces2
¡Hola desde los interfaces Java!
```

Observe que las implementaciones parciales pueden ser útiles, como cuando desea especificar en qué cantidad (pero no completa) deberían trabajar los métodos de una interfaz en una serie de subclases.

Crear un archivo JAR

"Vaya", dice el programador novato, "mis archivos de clase están aumentando de tamaño y hay muchos. ¿Tiene una idea de lo que tardará en descargarse mi applet?" Sonreímos y decimos, "Claro; es por esto por lo que debería utilizar archivos JAR". "¿*Qué* archivos?", pregunta al PN.

Un archivo Java (JAR) puede contener varios archivos. De hecho, los archivos JAR comprimen sus contenidos utilizando el formato ZIP, por lo que si su applet necesita una gran cantidad de archivos grandes, es conveniente crear un archivo JAR. Los navegadores de la red Internet únicamente necesitan una conexión en vez de nuevas conexiones para cada nuevo archivo, lo que puede mejorar los tiempos de descarga. También puede firmar digitalmente los archivos de un archivo JAR para probar su origen.

Creará archivos JAR con la herramienta jar y examinaremos esa herramienta en esta y las siguientes secciones. Esta herramienta viene con Java y aquí tiene su forma de uso:

```
jar [options] [manifest] destination input-file[additional input files]
```

Aquí, ***options*** son las opciones que puede utilizar con la herramienta JAR y son parecidas a las opciones que se utilizan con la herramienta UNIX tar. El argumento opcional ***manifest*** consiste en el nombre de un archivo *manifest*, que soporta la firma digital y muestra los contenidos del archivo Java. Cuando cree un JavaBean, utilice un archivo *manifest* para indicar qué clases son Beans (como veremos posteriormente en este capítulo).

Aquí tiene las opciones posibles que puede utilizar cuando use la herramienta jar:

- ***c***. Crea un archivo nuevo o vacío en la salida estándar.
- ***t***. Muestra la tabla de contenidos en la salida estándar.
- ***x file***. Extrae todos los archivos o sólo los nombres de archivos. Si file se omite, se extraen todos los archivos; en cualquier otro caso, únicamente se extraen los archivos especificados.
- ***f***. El segundo argumento especifica un archivo JAR para trabajar.
- ***v***. Genera una salida "duplicada" en stderr.
- ***m***. Incluye información manifiesta de un archivo manifest especificado.
- ***O***. Indica "sólo almacenar", sin utilizar compresión ZIP.
- ***M***. Especifica que un archivo *manifest* no debería crearse por las entradas.
- ***u***. Actualiza un archivo JAR existente añadiendo o cambiando los archivos del *manifest*.
- ***-C***. Cambia los directorios durante la ejecución del comando jar. Por ejemplo, jar añadiría todos los archivos dentro del directorio clases, pero no el propio directorio clases, al archivo jarfile.jar.



Truco: Puede utilizar un argumento comenzando por el carácter "@" para especificar un archivo que contiene argumentos adicionales, con un argumento por línea. Estos argumentos se insertan en la línea de comandos en la posición del argumento @filename.

Aquí tiene el uso típico de la herramienta jar:

```
c:\> jar cf jarfile.jar *.class
```

En este caso, todos los archivos de clase del directorio actual se sitúan en el archivo llamado jarfile.jar. La herramienta JAR genera automáticamente

un archivo *manifest* predeterminado y es siempre la primera entrada del archivo Java (por defecto, se llama META-INF/MANIFEST.MF).

Si tiene un archivo *manifest* y quiere que utilice la herramienta jar para un nuevo archivo JAR, puede utilizar la opción -m y especificarlo como sigue:

```
c:\> jar cmf manifest.mft jarfile.jar *.class
```

Observe que cuando especifica las opciones cfm en lugar de cmf, necesita especificar el nombre del archivo JAR primero, seguido del nombre del archivo *manifest*:

```
c:\> jar cfm jarfile.jar manifest.mft *.class
```

Observe también que los archivos JAR no son únicamente para archivos de clase; pueden almacenar cualquier tipo de archivo. Aquí tiene la forma de empaquetar todos los archivos de un directorio en un archivo JAR:

```
c:\> jar cfm jarfile.jar manifest.mft *.*
```

Si los nombres de archivo que quiere empaquetar incluían directorios, los directorios se buscan de forma recursiva y se almacenan en el JAR los archivos. Cuando el archivo JAR se desempaqueta, se recrea de nuevo la estructura de directorios.

Obtener los contenidos del archivo JAR

"Oh, oh", dice el programador novato, "tengo un archivo JAR y he olvidado el contenido. Supongo que tendré que crearlo de nuevo". "En absoluto", decimos, "basta con utilizar las opciones tf". "¿Cómo es eso?", pregunta el PN.

Puede determinar el contenido de un archivo JAR con las opciones tf, como en este caso, donde el archivo JAR contiene un *manifest* predeterminado en el directorio interno META-INF y varios archivos de clase:

```
c:\> jar tf jarfile.jar
META-INF/
META-INF/MANIFEST.MF
applet.class
jpanel.class
newButton.class
testPanel.class
```

Extraer archivos desde un archivo JAR

"Bien", dice el programador novato, "he utilizado archivos JAR para archivar mis archivos; pero, ¿cómo extraigo los archivos de un JAR?" Decimos, "Con las opciones xf". "¡Genial!", dice el PN.

Podemos extraer archivos desde un archivo JAR utilizando las opciones xf. Por ejemplo, aquí tiene la forma de extraer todos los archivos del archivo jarfile.jar introducido en el tema previo:

```
c:\> jar xf jarfile.jar
```

Al desempaquetar todos los archivos del archivo JAR de esta forma, también creamos su estructura original de directorio. En este caso, la herramienta jar desempaqueta los archivos de clase y también crea un directorio llamado META-INF donde coloca el archivo predeterminado *manifest* MANIFEST.MF.

También podemos extraer archivos especificando sus nombres. Aquí tiene un ejemplo en el que extraemos applet.class:

```
c:\> jar xf jarfile.jar applet.class
```

Apéndice. Contenido del CD-ROM

El CD-ROM que acompaña a este libro contiene elementos seleccionados específicamente para hacerlo aún más útil. En él encontrará:

- Una versión de prueba de Jbuilder con compatibilidad para Java 2.0 (PC).
- Una versión de prueba de CoffeeCup HTML Editor, que es un editor HTML compatible con Java y GIFs animados (PC/Linux).
- Formlayout. Un programa de diseño Java para construir formularios profesionales (PC).
- Una versión de prueba de JdesignerPro, que es un sistema integrado con todo lo necesario para desarrollar y distribuir aplicaciones basadas en exploradores Java (PC/Linux).
- Una versión de prueba de SuperMojo, que es la única herramienta RAD escrita íntegramente en Java que permite crear aplicaciones y JavaBeans (PC).
- Una versión de prueba de Ulead Photoimpact. Se trata de un programa de manipulación de imágenes capaz de crear, colorear, corregir y retocar imágenes fotográficas (PC).

- Una versión de prueba de Ulead Web Razor Pro. Un paquete que contiene cuatro estupendas utilidades de optimización, animación, diseño 3D y administración de imágenes. Web Razor Pro incluye GIF Animator 3.0, SmartSaver Pro, COOL 3D 2.0 y Photoexplorer (PC).

Requisitos del sistema

Software

- Su sistema operativo debe ser Windows 95, 98, NT4 o superior.
- Java Development Kit (JDK), version 1.2.2.
- Un editor de textos.
- Beans Development Kit (BDK).
- Java Servlet Development Kit (JSDK).

Hardware

- Un procesador 486DX o más rápido.
- El requisito mínimo de memoria RAM es 32Mb, pero es recomendable utilizar 48Mb.
- 65Mb de espacio libre de almacenamiento en disco.

Índice alfabético

&& y || lógicos, 151

A

Abandonar
- la herencia con final, 252
- la sobrescritura con final, 251

Abstract
- Button: base de los botones Swing, 583
- Windowing Toolkit, 266, 270

Acceso a
- los miembros sobrescritos, 242
- variables, 190

Actualización de barras de progreso, 684

Adición
- de imágenes a cuadros combinados, 676
- de imágenes a menús, 772

Adquirir e instalar Java, 37

Ajustar la alineación del campo de texto, 886

Ajuste de la extensión del deslizador, 639

Añadir
- botones y otros controles a menús, 790
- controles a las applets: botones, 300
- controles a las applets: Campos de texto, 297
- cuadros combinados y otros controles a barras de herramientas, 801
- e insertar utilizando StringBuffers, 130
- marcas de activación a menús, 505
- separadores de menú, 503
- y eliminar elementos de menú en tiempo de ejecución, 788

Aplicaciones, 269
- que se pueden ejecutar como applets, 317

Applets, 267

Árboles, 852

Áreas de texto, 372, 374

Argumentos de la línea de comandos pasados a main, 198

Arquitectura Modelo-Vista-Controlador, 530

Arrays, 82

Asignar la prioridad y detención de hilos, 920

B

Barras
- de desplazamiento, 373, 396, 614
- de desplazamiento y border layouts, 404

de herramientas, 758
de progreso, 664
Border Layout, 351
Borrar texto en StringBuffers, 131
Botones, 322, 329, 572, 589
de opción, 322, 339, 604
por defecto y mnemónicos, 595
toggle, 572, 597
Bucle
do-while, 167
for, 168
while, 164
anidado, 171
buscar clases Java con CLASSPATH, 66
Buscar y seleccionar texto en áreas de texto, 379
Búsqueda y reemplazamientos en cadenas, 122
bytecodes, 32

C

Cadenas, 85
Cambio de
mayúsculas a minúsculas (o viceversa) en cadenas, 123
tamaño automático de separadores, 693
Canvases, 454
Características Swing, 528
Card Layouts, 355
Casillas de activación, 322, 334, 601
y botones de opción, 572
Casting a nuevos tipos de datos, 101
Cerrar ventanas JFrame, 553
Clase
File, 895
JPanel, 546
MediaTracker, 458
Object de Java, 255
String, 110
StringBufferr, 124
Window, 484
Clases, 179
adaptador, 309
adaptador internas anónimas, 311
delegadas, 304
Foundation de Java, 523
Colección garbage
y el método finalize, 213
y gestión de memoria, 210
comandos de acción, 306
Comparación de cadenas, 158
Compilación, 51
revisión de los métodos censurados, 55
utilizando opciones en la línea de comandos, 52
Comprobar que un hilo está vivo, 919
Comunicación entre hilos, 927
Concatenación de cadenas, 120
Condicionales, 137
Configuración
de la orientación del panel de separación, 742
del tamaño del divisor de un panel de separación, 744
consolas de Java en los browsers, 297
content panes, 545
Convertir
automáticamente, 100
imágenes a escala de grises, 467
tipos de datos, 100
Creación de
aceleradores y mnemónicos de menú, 782
aplicaciones ventana, 72
applets, 69, 284
árboles, 871
arrays multidimensionales, 106, 109
arrays unidimensionales, 104
ayudas de herramientas, 687
barras de desplazamiento, 640
barras de herramientas, 797
barras de progreso, 679

cadenas, 117
campos de palabra clave, 890
campos de texto, 885
clases abstractas, 249
clases internas, 261
clases internas anónimas, 262
componentes de texto en Swing: la clase JTextComponent, 885
constantes con final, 253
constructores, 204
cuadros combinados, 665
  editables, 674
cuadros de diálogo
  con JDialog, 843
  con panel de opciones de campo de texto de entrada, 837
  con panel de opciones de confirmación, 834
  con panel de opciones de marcos internos, 841
  con panel de opciones de mensaje, 835
  con panel de opciones para entrada de cuadros combinados, 839
deslizadores, 630
elementos de menú de casillas de verificación, 774
filtros para selectores de archivo, 709
gráficos animados con hilos, 931
grupos de botones toggle, 600
hilos múltiples, 916
listas, 646
literales
  boleanos, 91
  carácter, 91
  en coma flotante, 89
  enteros, 88
  tipo cadena, 93
marcos internos, 814
menús, 489
  de botones de activación, 777
  emergentes, 791
métodos, 193
  de acceso a datos, 202
  de clase, 201
multiniveles de herencia, 237
objetos
  Menu, 493
  MenuItem, 495
paneles de desplazamiento, 621
  con cabeceras y bordes, 627
paneles de lengüetas, 725
paquetes que contienen paquetes, 943
selectores de archivos, 699
separadores, 690
StringBuffers, 125
submenús, 508, 780
tablas, 852
un applet Swing, 545
un archivo JAR, 946
un elemento de menú, 765
un hilo con la clase Thread, 911
un hilo con la interfaz Runnable, 909
un menú, 761
un modelo de cuadro combinado, 678
un modelo de lista personalizado, 657
un objeto MenuBar, 491
un panel de escritorio, 812
un paquete, 941
un renderizador personalizado
  para celdas de lista, 657
  para el cuadro combinado, 678
un selector de color, 695
un sistema de menús básico, 768
una aplicación Swing, 549
una barra de menús, 758
una interfaz, 944
una subclase, 229
una ventana, 806
una ventana de marco, 810
variables de clase, 191
variables de instancia, 188

ventanas de aplicación, 312
ventanas Frame, 475
Cuadros
combinados, 663
de desplazamiento, 407
de diálogo, 475, 512, 806
de lista desplegables, 372, 391
de texto, 321, 324, 580

D

Dar brillo a las imágenes, 466
Dar nombre a un hilo, 907
Declaración de arrays
multidimensionales, 105
unidimensionales, 103
Declaración de variables de tipo
booleano, 96
carácter, 95
coma flotante, 94
entero, 93
Declaración y creación de objetos, 182
Declarar y definir clases, 186
Deshabilitar elementos de menú, 503
Deslizadores, 614
Desplazamiento de imágenes, 629
Desplazar campos de texto, 889
Detener un hilo, 908
Devolver
arrays desde métodos, 222
objetos desde métodos, 221
valores desde los métodos, 199
Dibujar
arcos, 450
gráficos, 439
en applets, 293
óvalos, 447
polígonos, 450
rectángulos, 447
redondeados, 448
rectas, 447
Dibujo libre, 449
Diseño de programas Java, 74
Dispatching dinámico de métodos (Polimorfismo en tiempo de ejecución), 246
Disponibilidad, 76
Distribución del programa Java, 77
Doble buffer, 937

E

Ejecutar
aplicaciones ventana, 73
applets, 71
Eliminar el parpadeo en animaciones gráficas, 934
Escalas if-else, 161
Escribir código
conocer las palabras reservadas de Java, 45
creación de archivos de código, 44
crear una aplicación, 48
Especificadores de acceso y herencia, 231
Espera (para unión) de hilos, 917
Esquemas de flujo (flow layout), 341
Establecer
caracteres en StringBuffers, 130
el acceso a los métodos, 195
la apariencia, 560
los componentes para la apariencia, 565
los modos de dibujo, 450
etiqueta HTML <APPLET>, 287
etiquetas, 327, 573
y cuadros de texto, 571
eventos de barras de progreso, 686
eventos de selección del cuadro combinado, 672
Evitar las referencias circulares, 212
Extender componentes, 307
Extensibilidad, 76
Extraer archivos desde un archivo JAR, 948

F

forma antigua de gestionar eventos, 307
Formateo de números en cadenas, 124
fuentes, 425

G

Gestión
  de browsers no Java, 290
  de eventos, 300
  de eventos de ventana, 480
  de eventos estándar, 302
  de los eventos de menú, 498
  de multiniveles de constructores, 239
  de selecciones múltiples, 653
gestor
  de distribución de cuadro, 745
  de distribución de superposición, 752
Gráficos, 415
Grid bag layouts, 358
Grid layouts, 345

H

Habilitar/inhabilitar elementos de menú, 785
Herencia, 182
Herramientas de ayuda, 665
hilos, 905

I

iconos imagen, 577
If anidados, 161
Imágenes, 415, 434
  en casillas de activación y botones de opción, 607
  en etiquetas, 579
  rollover y deshabilitadas, 594
Implementación parcial de una interfaz, 945
importando paquetes y clases Java, 64
Incremento y decremento: ++ y --, 141
Inicialización
  de arrays
    multidimensionales, 108
    unidimensionales, 105
  de variables, 98
  dinámica, 99
Insets, 555
intercalados y rellenos, 365
interfaces para herencia múltiple, 258
interfaz ImageObserver, 456
Introducir etiquetas <APPLET> en el código, 290

J

Java, 30
JOptionPane para crear cuadros de diálogo, 825

L

layered panes, 542
Leer parámetros en applets, 295
listas de selección múltiple, 388
Listas, 371, 381, 614
Longitud de un array, 110

M

Mantenimiento, 75
Menús, 474, 757
  emergentes, 510
Métodos, 181
  init, start, stop, destroy, paint y update, 291
  sincronizados, 926
Miembros de datos, 180
Modos de selección de lista, 653
Módulo: %, 145
Mostrar y ocultar ventanas, 477
Multiplicación y división: * y /, 144

N

NOT unario: ~ y !, 143

O

Objetos, 179
Obtención
de caracteres y substring, 121
de entrada de los cuadros de diálogo creados con JDialog, 848
de la longitud de la cadena, 119
de los contenidos del archivo JAR, 948
del hilo principal, 907
y establecimiento de longitudes y capacidades de StringBuffer, 129
Obtener y fijar el estado de las casillas de activación y de los botones de opción, 608
Ocultar ventanas automáticamente al cerrarlas, 483
Opciones de compilación cruzada, 55
operador If-Then-Else: ?:, 153
Operadores, 135
de asignación: = y [operador]=, 154
de desplazamiento: >>, >>> y <<, 146
de relación:>, >=, <, <=, == y !=, 147
lógicos a nivel de bit AND, Xor y OR: &, ^ y |, 148
Orígenes del lenguaje Java, 32
Otros recursos, 26

P

palabra clave this, 220
Paneles, 349
de capas, 716, 722
de desplazamiento, 373, 614
de lengüetas, 716
de separación, 716, 734
expandibles con un clic, 740
Pasar
arrays a métodos, 219
objetos a métodos, 217
parámetros a constructores, 205
parámetros a los métodos, 196
Pintar
en Swing frente a AWT, 545
etiquetas en un deslizador, 638
las marcas de un deslizador, 637
Plantillas, 323
plug-in de Java del browser, 294
Precedencia de operadores, 140
Preparar para crear un applet Swing, 536
Procesamiento de doble clic en listas, 658
Programas Java, 34
public class app, 49

R

ratón, 416
Realzar imágenes, 469
Recursividad, 209
Redimensionar imágenes, 437
Reemplazar texto
en áreas de texto, 377
en StringBuffers, 132
Relación es-a frente a tiene-a, 253
Relleno de un deslizador, 636
Rendimiento, 75
Requerimientos, 25
root panes, 539

S

Salir de una aplicación al cerrar su ventana, 316
seguridad del lenguaje Java, 33
Seleccionar
colores, 450
los bordes del componente, 555
Selectores, 664
Sentencia
break, 172
continue, 174

else, 160
if, 159
switch, 162
Separadores, 665
Sincronizar bloques de código, 924
Sobrecarga
de constructores, 215
de métodos, 214
Sobrescritura de métodos, 241
Suma y resta: + y -, 145
Suspender
y reanudar animaciones gráficas, 935
y reanudar hilos, 929
Swing, 524
y el orden Z, 717

T

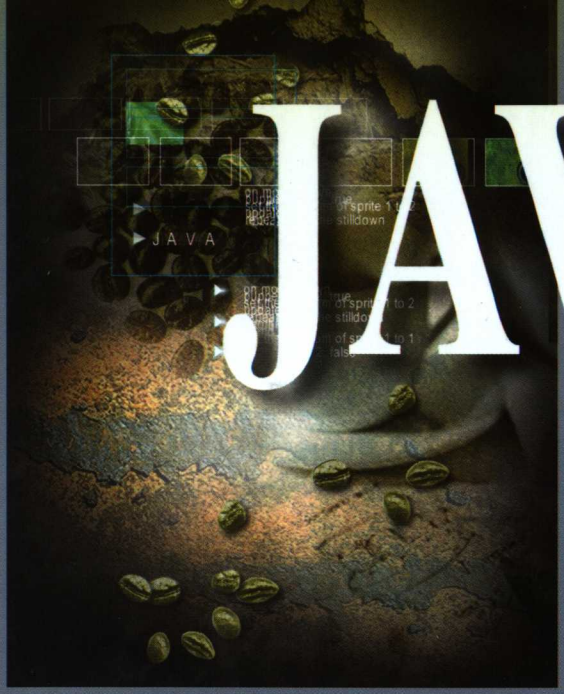
Tablas, 851
teclado, 420
y ratón, 416
Texto y fuentes, 416
Tipos de datos, 81
Trabajar con
ByteArrayInputStream, 900
ByteArrayOutputStream, 901
FileInputStream, 897
FileOutputStream, 899
InputStream, 897
OutputStream, 897
Swing, 531
Trabajar pixel por pixel, 462
Transparencia en los componentes de la Swing, 719

U

Utilización de la clase Math, 156
Utilizar paneles en la programación de gráficos, 529

V

variables de superclase con objetos de subclases, 244
Variables, 79
Ventanas, 473, 805
viewports, 615, 613
Visualización de imágenes en listas, 655
Visualizar
controles en Swing frente a AWT, 546
imágenes en botones, 592



JAVA 2

Aprenda a dominar Java en profundidad.

- Comprenda el Kit de Herramientas de Gestión de Ventanas o *Abstrac Windowing Tollking (AWT).*
- Cree Java Beans.
- Maneje hilos múltiples.
- Domine *sockets* y la gestión de red.
- Cree clientes y servidores TCP.
- Utilice el Paquete de Conectividad de Bases de datos (JDBC).
- Cree paquetes, interfaces y archivos JAR.
- Descubra los temas de seguridad.

La biblia de Java 2 abarca desde la sintaxis más básica a la programación más avanzada para la red Internet. Los temas incluyen la seguridad de la red Internet, cobertura de todos los componentes Swing de Java y escritura bidireccional de programas de gestión de la red Internet. Además, cada tema se ilustra al menos con un ejemplo completo.

Creado por el experto programador y autor de super ventas, Steve Holzner, La biblia de Java 2 es la compilación más impresionante de Java, desde los temas básicos hasta los más avanzados, en un profundo libro de referencia.

INCLUYE CD-ROM



con el código, imágenes y otros ficheros utilizados en el libro, así como un gran número de utilidades.

en línea
Servicio de Información en Internet
http://www.en-linea.net

ANAYA MULTIMEDIA

NIVELES
- Iniciación
- Básico
- ✓ Medio
- ✓ Avanzado

TIPO DE LIBRO
- Referencia

TEMÁTICA
- Programación (Internet)

2310223
ISBN 84-415-1037-7
8441510371