Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

Componentes de la GUI: parte 1

12

¿Crees que puedo escuchar todo el día esas cosas?

——Lewis Carroll

Incluso un evento menor en la vida de un niño es un evento del mundo de ese niño y, por ende, es un evento mundial.

——Gastón Bachelard

Tú pagas, por lo tanto, tú decides.

——Punch

Objetivos

En este capítulo aprenderá:

- A usar la apariencia visual de Nimbus.
- A crear interfaces gráficas de usuario (GUI) y a manejar los eventos generados por las interacciones de los usuarios con las GUI.
- Los paquetes que contienen componentes relacionados con las GUI, las clases y las interfaces manejadoras de eventos.
- A crear y manipular botones, etiquetas, listas, campos de texto y paneles.
- A manejar los eventos de ratón y los eventos de teclado.
- A utilizar los administradores de esquemas para ordenar los componentes de las GUI.

Plan general

12.1 Introducción

Una **interfaz gráfica de usuario** (**GUI**) ofrece un mecanismo amigable para que el usuario interactúe con una aplicación. Una GUI dota a una aplicación con una "apariencia visual" única. Las GUI se construyen a partir de los **componentes de GUI**. Con frecuencia a éstos se les denomina *controles* o *widgets* (abreviación para gadgets de ventana). Un componente de GUI es un objeto con el que el usuario *interactúa* a través del ratón, del teclado o de otra forma de entrada, como el reconocimiento de voz. En este capítulo y en el capítulo 22, GUI Components: Part 2, aprenderá sobre muchos de los llamados **componentes GUI de Swing** del paquete `javax.swing`. A medida que se vayan requiriendo, a lo largo del libro cubriremos otros componentes de GUI. En el capítulo 25 y en dos capítulos en línea aprenderá sobre JavaFX, que son las API para GUI, gráficos y multimedia más recientes de Java.

Observación de apariencia visual 12.1

Al dotar a distintas aplicaciones con componentes de interfaz de usuario consistentes e intuitivos, los usuarios pueden familiarizarse en cierto modo con una nueva aplicación, de manera que pueden aprender a utilizarla en menor tiempo y con mayor productividad.

Soporte de IDE para el diseño de GUI

Muchos IDE cuentan con herramientas de diseño de GUI, con las que podemos especificar visualmente el *tamaño*, la *ubicación* y otros atributos de un componente, ya sea mediante el ratón, el teclado o la técnica de "arrastrar y soltar". Los IDE generan el código de GUI por usted. Aunque esto simplifica en gran medida la creación de GUI, cada IDE genera este código de manera distinta. Por esta razón, escribimos el código de GUI a mano, como verá en los archivos de código fuente de los ejemplos de este capítulo. Le recomendamos que cree cada GUI en forma visual, mediante el uso del IDE de su preferencia.

GUI de ejemplo: la aplicación de demostración `SwingSet3`

Como ejemplo de una GUI, en la figura 12.1 se muestra una aplicación demostrativa de **SwingSet3**, descargada de la sección JDK 8 Demos and Samples en `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. Esta aplicación presenta una forma conveniente de que usted explore los diversos componentes de GUI que ofrecen las API de la GUI de Swing de Java. Sólo haga clic en el nombre de un componente (como `JFrame`, `JTabbedPane`, etc.) en el área `GUI Components` de la parte izquierda de la pantalla y verá una demostración del componente de GUI en la sección derecha de la ventana. El código fuente de cada demostración se muestra en el área de texto de la parte inferior de la ventana. Hemos etiquetado algunos de los componentes de GUI en la aplicación. En la parte superior de la ventana hay una **barra de título**, la cual contiene el título de la ventana. Debajo de la barra de título hay una **barra de menús** que contiene **menús** (`File` y `View`). En la región superior derecha de la ventana hay un conjunto de **botones**; por lo general, los usuarios oprimen botones para realizar tareas. En el área `GUI Components` de la ventana, hay un **cuadro combinado**; el usuario puede hacer clic en la flecha hacia abajo que está al lado derecho del cuadro para seleccionar un elemento de la lista. Los menús, botones y el cuadro combinado son parte de la GUI de la aplicación y nos permiten interactuar con ella.

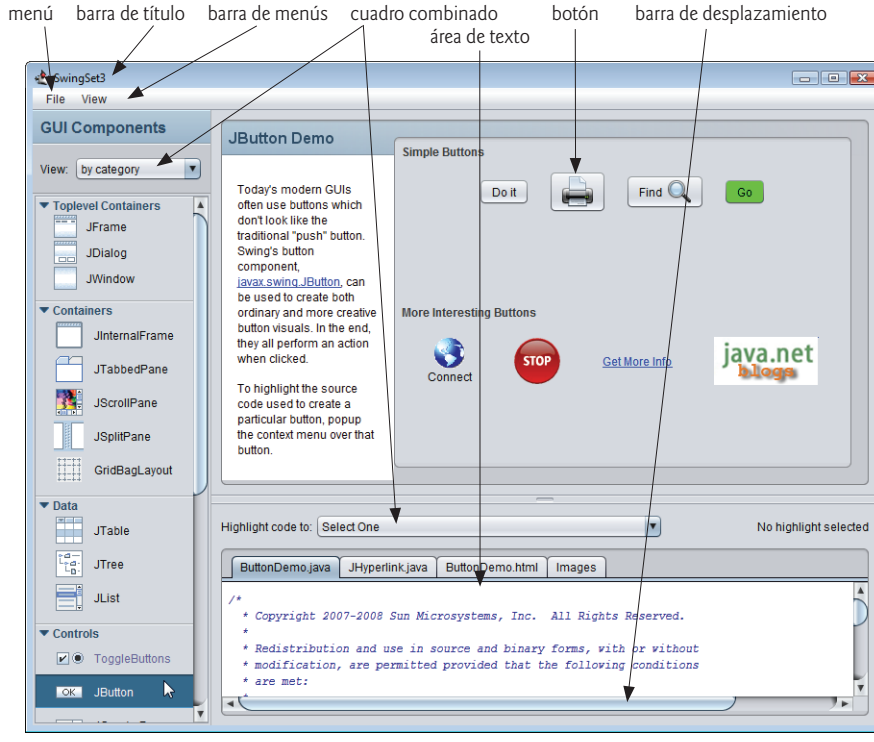


Fig. 12.1 | La aplicación **SwingSet3** demuestra muchos de los componentes de la GUI Swing de Java.

12.2 La apariencia visual Nimbus de Java

La apariencia de una GUI está compuesta por sus aspectos visuales, como sus colores y tipos de letra, mientras que su parte "tangible" consiste en los componentes que usa para interactuar con la GUI, como los botones y menús. En conjunto se conocen como la apariencia visual de la GUI o el *look-and-feel*. Swing

tiene una apariencia visual multiplataforma conocida como **Nimbus**. Para las capturas de pantalla de GUI, como la figura 12.1, configuramos nuestros sistemas para usar Nimbus como la apariencia visual predeterminada. Hay tres formas en que podemos usar Nimbus:

1. Establecerla como predeterminada para todas las aplicaciones Java que se ejecuten en la computadora.
2. Establecerla como la apariencia visual al momento de iniciar una aplicación, pasando un argumento de línea de comandos al comando `java`.
3. Establecerla como la apariencia visual mediante programación en nuestra aplicación (vea la sección 22.6).

Para establecer a Nimbus como predeterminada para todas las aplicaciones Java, es necesario crear un archivo de texto llamado `swing.properties` en la carpeta `lib` de su carpeta de instalación del JDK y del JRE. Coloque la siguiente línea de código en el archivo:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Además del JRE independiente, hay un JRE anidado en su carpeta de instalación del JDK. Si utiliza un IDE que dependa del JDK, tal vez también necesite colocar el archivo `swing.properties` en la carpeta `lib` de la carpeta `jre` anidada.

Si prefiere seleccionar Nimbus en cada aplicación individual, coloque el siguiente argumento de línea de comandos después del comando `java` y antes del nombre de la aplicación al momento de ejecutarla:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

12.3 Entrada/salida simple basada en GUI con `JOptionPane`

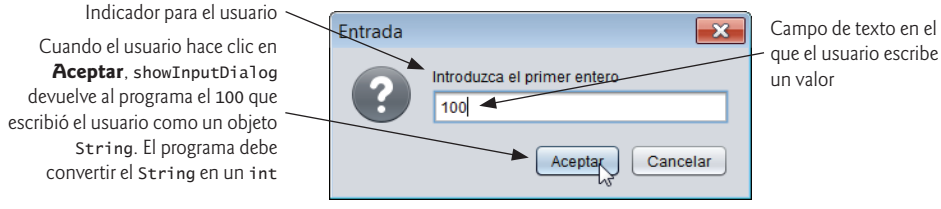
Las aplicaciones en los capítulos 2 a 10 muestran texto en la ventana de comandos y obtienen la entrada de la ventana de comandos. La mayoría de las aplicaciones que usamos a diario utilizan ventanas o **cuadros de diálogo** (también conocidos como **diálogos**) para interactuar con el usuario. Por ejemplo, los programas de correo electrónico le permiten escribir y leer mensajes en una ventana que proporciona el programa. Los cuadros de diálogo son ventanas en las cuales los programas muestran mensajes importantes al usuario, u obtienen información de éste. La clase **`JOptionPane`** de Java (paquete `javax.swing`) proporciona cuadros de diálogo prefabricados para entrada y salida. Estos diálogos se muestran mediante la invocación de los métodos `static` de `JOptionPane`. La figura 12.2 presenta una aplicación simple de suma, que utiliza dos **diálogos de entrada** para obtener enteros del usuario, y un **diálogo de mensaje** para mostrar la suma de los enteros que introduce el usuario.

```
1  // Fig. 12.2: Suma.java
2  // Programa de suma que utiliza a JOptionPane para entrada y salida.
3  import javax.swing.JOptionPane;
4
5  public class Suma
6  {
7     public static void main(String[] args)
8     {
9        // obtiene la entrada del usuario de los diálogos de entrada de JOptionPane
10       String primerNumero =
11          JOptionPane.showInputDialog("Introduzca el primer entero");
```

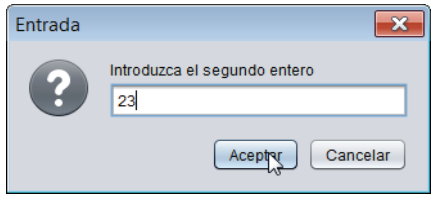
Fig. 12.2 | Programa de suma que utiliza a `JOptionPane` para entrada y salida (parte 1 de 2).

```
12          String segundoNumero =
13              JOptionPane.showInputDialog("Introduzca el segundo entero");
14
15          // convierte las entradas String en valores int para usarlos en un cálculo
16          int numero1 = Integer.parseInt(primerNumero);
17          int numero2 = Integer.parseInt(segundoNumero);
18
19          int suma = numero1 + numero2;
20
21          // muestra los resultados en un diálogo de mensajes de JOptionPane
22          JOptionPane.showMessageDialog(null, "La suma es " + suma,
23             "Suma de dos enteros", JOptionPane.PLAIN_MESSAGE);
24       }
25    } // fin de la clase Suma
```

(a) Diálogo de entrada mostrado por las líneas 10 y 11



(b) Diálogo de entrada mostrado por las líneas 12 y 13



(c) Diálogo de mensaje mostrado por las líneas 22 y 23; Cuando el usuario hace clic en **Aceptar**, el diálogo de mensaje se cierra (se quita de la pantalla)

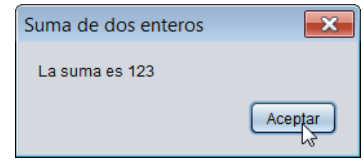


Fig. 12.2 | Programa de suma que utiliza a `JOptionPane` para entrada y salida (parte 2 de 2).

Diálogos de entrada

La línea 3 importa la clase `JOptionPane`. Las líneas 10 y 11 declaran la variable `String primerNumero`, y le asignan el resultado de la llamada al método `static` **`showInputDialog`** de `JOptionPane`. Este método muestra un diálogo de entrada (vea la primera captura de pantalla en la figura 12.2(a)), usando el argumento `String` del método (`"Introduzca el primer entero"`) como indicador.

Observación de apariencia visual 12.2

El indicador en un diálogo de entrada utiliza comúnmente las ***mayúsculas y minúsculas estilo oración****, que pone en mayúscula sólo la primera letra de la primera palabra en el texto, a menos que la palabra sea un nombre propio (por ejemplo, Jones).*

El usuario escribe caracteres en el campo de texto y después hace clic en el botón **Aceptar** u oprime la tecla *Intro* para enviar el objeto `String` al programa. Al hacer clic en **Aceptar** también se **cierra (oculta) el diálogo**. [*Nota*: si escribe en el campo de texto y no aparece nada, actívelo haciendo clic sobre él con el ratón]. A diferencia de `Scanner`, que puede utilizarse para que el usuario introduzca valores de *varios* tipos mediante el teclado, *un diálogo de entrada sólo puede introducir objetos* `String`. Esto es común

en la mayoría de los componentes de la GUI. El usuario puede escribir *cualquier* carácter en el campo de texto del diálogo de entrada. Nuestro programa asume que el usuario introduce un valor entero *válido*. Si el usuario hace clic en el botón **Cancelar**, `showInputDialog` devuelve `null`. Si el usuario escribe un valor no entero o si hace clic en el botón **Cancelar** en el diálogo de entrada, ocurrirá una excepción y el programa no operará en forma correcta. Las líneas 12 y 13 muestran otro diálogo de entrada que pide al usuario que introduzca el segundo entero. Cada diálogo `JOptionPane` que usted muestre en pantalla se conoce como **diálogo modal**. Mientras que el diálogo esté en la pantalla, el usuario *no podrá* interactuar con el resto de la aplicación.

Observación de apariencia visual 12.3

No haga uso excesivo de los diálogos modales, ya que pueden reducir la capacidad de uso de sus aplicaciones. Use un diálogo modal sólo cuando necesite evitar que los usuarios interactúen con el resto de una aplicación hasta que cierren el diálogo.

Convertir objetos `String` en valores `int`

Para realizar el cálculo, debemos convertir los objetos `String` que el usuario introdujo, en valores `int`. Recuerde que el método `static` `parseInt` de la clase `Integer` convierte su argumento `String` en un valor `int` y podría lanzar una excepción `NumberFormatException`. Las líneas 16 y 17 asignan los valores convertidos a las variables locales `numero1` y `numero2`. Después, la línea 19 suma estos valores.

Diálogos de mensaje

Las líneas 22 y 23 usan el método `static` **`showMessageDialog`** de `JOptionPane` para mostrar un diálogo de mensaje (la última captura de pantalla de la figura 12.2) que contiene la suma. El primer argumento ayuda a la aplicación de Java a determinar en dónde debe *colocar* el cuadro de diálogo. Por lo general, un diálogo se muestra desde una aplicación GUI con su propia ventana. El primer argumento se refiere a esa ventana (la cual se denomina *ventana padre*) y hace que el diálogo aparezca centrado sobre el padre (como veremos en la sección 12.9). Si el primer argumento es `null`, el cuadro de diálogo se muestra en la parte *central* de la pantalla. El segundo argumento es el *mensaje* a mostrar; en este caso, el resultado de concatenar el objeto `String` `"La suma es "` y el valor de `suma`. El tercer argumento (`"Suma de dos enteros"`) representa el objeto `String` que debe aparecer en la *barra de título* del diálogo, en la parte superior. El cuarto argumento (**`JOptionPane.PLAIN_MESSAGE`**) es el *tipo de diálogo de mensaje a mostrar*. Un diálogo `PLAIN_MESSAGE` *no* muestra un *icono* a la izquierda del mensaje. La clase `JOptionPane` proporciona varias versiones sobrecargadas de los métodos `showInputDialog` y `showMessageDialog`, así como métodos que muestran otros tipos de diálogos. Para obtener la información completa, visite el sitio `http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html`.

Observación de apariencia visual 12.4

*Por lo general, la barra de título de una ventana utiliza **mayúsculas y minúsculas como en el título de libro**: un estilo que usa mayúscula para la primera letra de cada palabra significativa en el texto, y no termina con ningún signo de puntuación (por ejemplo, Mayúsculas en el Título de un Libro).*

Constantes de diálogos de mensajes de `JOptionPane`

Las constantes que representan los tipos de diálogos de mensajes se muestran en la figura 12.3. Todos los tipos de diálogos de mensaje, excepto `PLAIN_MESSAGE`, muestran un icono a la *izquierda* del mensaje. Estos iconos proporcionan una indicación visual de la importancia del mensaje para el usuario. Un icono `QUESTION_MESSAGE` es el *icono predeterminado* para un cuadro de diálogo de entrada (vea la figura 12.2).

Tipo de diálogo de mensaje	Icono	Descripción
`ERROR_MESSAGE`		Indica un error.
`INFORMATION_MESSAGE`		Indica un mensaje informativo.
`WARNING_MESSAGE`		Advierte al usuario sobre un problema potencial.
`QUESTION_MESSAGE`		Hace una pregunta al usuario. Por lo general, este diálogo requiere una respuesta, como hacer clic en un botón **Sí** o **No**.
`PLAIN_MESSAGE`	sin icono	Un diálogo que contiene un mensaje, pero no un icono.

Fig. 12.3 | Constantes `static` de `JOptionPane` para diálogos de mensaje.

12.4 Generalidades de los componentes de Swing

Aunque es posible realizar operaciones de entrada y salida utilizando los diálogos de `JOptionPane`, la mayoría de las aplicaciones de GUI requieren interfaces de usuario más elaboradas. El resto de este capítulo habla acerca de muchos componentes de la GUI que permiten a los desarrolladores de aplicaciones crear GUI robustas. La figura 12.4 lista varios de los componentes básicos de la GUI de Swing que vamos a analizar.

Componente	Descripción
`JLabel`	Muestra *texto* o iconos que *no pueden editarse*.
`JTextField`	Por lo general *recibe entrada* del usuario.
`JButton`	Activa un evento cuando se oprime mediante el ratón.
`JCheckBox`	Especifica una opción que puede *seleccionarse* o *no seleccionarse*.
`JComboBox`	Una *lista desplegable de elementos*, a partir de los cuales el *usuario* puede realizar una *selección*.
`JList`	Una *lista de elementos* a partir de los cuales el usuario puede realizar una *selección*, *haciendo clic* en *cualquiera* de ellos. *Pueden* seleccionarse *varios* elementos.
`JPanel`	Un área en la que pueden *colocarse* y *organizarse* los *componentes*.

Fig. 12.4 | Algunos componentes básicos de GUI.

Comparación entre Swing y AWT

En realidad, hay *dos* conjuntos de componentes de GUI en Java. En los primeros días de Java, las GUI se creaban a partir de componentes del **Abstract Window Toolkit (AWT)** en el paquete **`java.awt`**. Éstos se ven como los componentes de GUI nativos de la plataforma en la que se ejecuta un programa de Java. Por ejemplo, un objeto de tipo `Button` que se muestra en un programa de Java ejecutándose en Microsoft Windows tendrá la misma apariencia que los botones en las demás aplicaciones *Windows*. En el sistema operativo Apple Mac OS X, el objeto `Button` tendrá la misma apariencia visual que los botones en las demás aplicaciones *Mac*. Algunas veces, incluso la forma en la que un usuario puede interactuar con un componente específico del AWT *difiere entre una plataforma y otra*. A la apariencia y la forma en la que interactúa el usuario con la aplicación se conoce como su **apariencia visual**.

Observación de apariencia visual 12.5

Los componentes de la GUI de Swing nos permiten especificar una apariencia visual uniforme para nuestras aplicaciones en todas las plataformas, o usar la apariencia visual personalizada en cada plataforma. Una aplicación puede incluso cambiar la apariencia visual durante la ejecución, para permitir a los usuarios elegir su propia apariencia visual preferida.

Comparación entre componentes de GUI ligeros y pesados

La mayoría de los componentes de Swing son **componentes ligeros**; es decir, que se escriben, manipulan y visualizan por completo en Java. Los componentes de AWT son **componentes pesados**, ya que dependen del **sistema de ventanas** de la plataforma local para determinar su funcionalidad y su apariencia visual. Varios componentes de Swing son *pesados*.

Superclases de los componentes de GUI ligeros de Swing

El diagrama de clases de UML de la figura 12.5 muestra una *jerarquía de herencia* de clases, a partir de las cuales los componentes ligeros de Swing heredan sus atributos y comportamientos comunes.

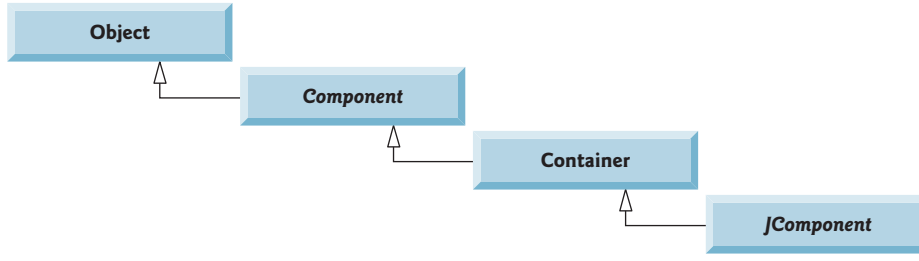


Fig. 12.5 | Superclases comunes de los componentes ligeros de Swing.

La clase `Component` (paquete `java.awt`) es una superclase que declara las características comunes de los componentes de GUI en los paquetes `java.awt` y `javax.swing`. Cualquier objeto que *sea un* `Container` (paquete `java.awt`) se puede utilizar para organizar a otros objetos `Component`, *adjuntando* esos objetos `Component` al objeto `Container`. Los objetos `Container` se pueden colocar en otros objetos `Container` para organizar una GUI.

La clase `JComponent` (paquete `javax.swing`) es una subclase de `Container`. `JComponent` es la superclase de todos los componentes *ligeros* de Swing, y declara los atributos y comportamientos comunes. Debido a que `JComponent` es una subclase de `Container`, todos los componentes ligeros de Swing son también objetos `Container`. Algunas de las características comunes que soporta `JComponent` son:

1. Una **apariencia visual adaptable**, la cual puede utilizarse para *personalizar* la apariencia de los componentes (por ejemplo, para usarlos en plataformas específicas). En la sección 22.6 veremos un ejemplo de esto.
2. Teclas de método abreviado (llamadas **nemónicos**) para un acceso directo a los componentes de la GUI por medio del teclado. En la sección 22.4 veremos un ejemplo de esto.
3. Breves descripciones del propósito de un componente de la GUI (lo que se conoce como **cuadros de información sobre herramientas** o ***tool tips***) que se muestran cuando el *cursor del ratón se coloca sobre el componente* durante un breve periodo. En la siguiente sección veremos un ejemplo de esto.
4. Soporte para *accesibilidad*, como lectores de pantalla Braille para las personas con impedimentos visuales.
5. Soporte para la **localización** de la interfaz de usuario; es decir, personalizar la interfaz de usuario para mostrarla en distintos lenguajes y utilizar las convenciones de la cultura local.

12.5 Presentación de texto e imágenes en una ventana

Nuestro siguiente ejemplo introduce un marco de trabajo para crear aplicaciones de GUI. Este marco de trabajo utiliza varios conceptos que aparecerán en muchas de nuestras aplicaciones de GUI. Éste es nuestro primer ejemplo en el que la aplicación aparece en su propia ventana. La mayoría de las ventanas que creará y que pueden contener componentes de GUI de Swing son una instancia de la clase `JFrame` o una subclase de `JFrame`. Ésta es una subclase *indirecta* de la clase `java.awt.Window` que proporciona los atributos y comportamientos básicos de una ventana, como una *barra de título* en la parte superior, y *botones* para *minimizar*, *maximizar* y *cerrar* la ventana. Como la GUI de una aplicación por lo general es específica para esa aplicación, la mayoría de nuestros ejemplos consistirán en *dos* clases: una subclase de `JFrame` que nos ayuda a demostrar los nuevos conceptos de la GUI y una clase de aplicación, en la que `main` crea y muestra la ventana principal de la aplicación.

Etiquetado de componentes de la GUI

Una GUI típica consiste en muchos componentes. Con frecuencia, los diseñadores de GUI proporcionan texto que indica el propósito de cada componente. Dicho texto se conoce como **etiqueta** y se crea con la clase **`JLabel`**, que es una subclase de `JComponent`. Un objeto `JLabel` muestra texto de sólo lectura, una imagen, o texto y una imagen. Raras veces las aplicaciones modifican el contenido de una etiqueta después de crearla.

Observación de apariencia visual 12.6

Por lo general, el texto en un objeto `JLabel` utiliza las mayúsculas y minúsculas estilo oración.

La aplicación de las figuras 12.6 y 12.7 demuestra varias características de `JLabel` y presenta la estructura que utilizamos en la mayoría de nuestros ejemplos de GUI. *No* resaltamos el código en este ejemplo, ya que casi todo es nuevo. [*Nota*: hay muchas más características para cada componente de GUI de las que podemos cubrir en nuestros ejemplos. Para conocer todos los detalles acerca de cada componente de la GUI, visite su página en la documentación en línea. Para la clase `JLabel`, visite `docs.oracle.com/javase/7/docs/api/javax/swing/JLabel.html`].

```
1  // Fig. 12.6: LabelFrame.java
2  // Componentes Jlabel con texto e iconos.
3  import java.awt.FlowLayout; // especifica cómo se van a ordenar los componentes
4  import javax.swing.JFrame; // proporciona las características básicas de una
                              ventana
5  import javax.swing.JLabel; // muestra texto e imágenes
6  import javax.swing.SwingConstants; // constantes comunes utilizadas con Swing
7  import javax.swing.Icon; // interfaz utilizada para manipular imágenes
8  import javax.swing.ImageIcon; // carga las imágenes
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel etiqueta1; // JLabel sólo con texto
13    private JLabel etiqueta2; // JLabel construida con texto y un icono
14    private JLabel etiqueta3; // JLabel con texto adicional e icono
15
16    // El constructor de LabelFrame agrega objetos JLabel a JFrame
17    public LabelFrame()
18    {
```

Fig. 12.6 | Componentes `JLabel` con texto e iconos (parte 1 de 2).

```
19         super("Prueba de JLabel");
20         setLayout(new FlowLayout()); // establece el esquema del marco
21
22         // Constructor de JLabel con un argumento String
23         etiqueta1 = new JLabel("Etiqueta con texto");
24         etiqueta1.setToolTipText("Esta es etiqueta1");
25         add(etiqueta1); // agrega etiqueta1 a JFrame
26
27         // Constructor de JLabel con argumentos de cadena, Icono y alineación
28         Icon insecto = new ImageIcon(getClass().getResource( "insecto1.png"));
29         etiqueta2 = new JLabel("Etiqueta con texto e icono", insecto,
30            SwingConstants.LEFT);
31         etiqueta2.setToolTipText("Esta es etiqueta2");
32         add(etiqueta2); // agrega etiqueta2 a JFrame
33
34         etiqueta3 = new JLabel(); // constructor de JLabel sin argumentos
35         etiqueta3.setText("Etiqueta con icono y texto en la parte inferior");
36         etiqueta3.setIcon(insecto); // agrega icono a JLabel
37         etiqueta3.setHorizontalTextPosition(SwingConstants.CENTER);
38         etiqueta3.setVerticalTextPosition(SwingConstants.BOTTOM);
39         etiqueta3.setToolTipText("Esta es etiqueta3");
40         add(etiqueta3); // agrega etiqueta3 a JFrame
41      }
42   } // fin de la clase LabelFrame
```

Fig. 12.6 | Componentes `JLabel` con texto e iconos (parte 2 de 2).

```
1  // Fig. 12.7: PruebaLabel.java
2  // Prueba de LabelFrame.
3  import javax.swing.JFrame;
4
5  public class PruebaLabel
6  {
7     public static void main(String[] args)
8     {
9        LabelFrame marcoEtiqueta = new LabelFrame();
10       marcoEtiqueta.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoEtiqueta.setSize(260, 180);
12       marcoEtiqueta.setVisible(true);
13    }
14 } // fin de la clase PruebaLabel
```

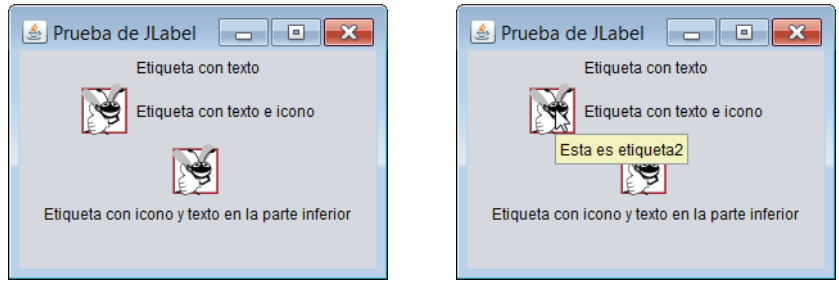


Fig. 12.7 | Prueba de `LabelFrame`.

La clase `LabelFrame` (figura 12.6) extiende a `JFrame` para heredar las características de una ventana. Utilizaremos una instancia de la clase `LabelFrame` para mostrar una ventana que contiene tres objetos `JLabel`. Las líneas 12 a 14 declaran las tres variables de instancia `JLabel`, cada una de las cuales se instancia en el constructor de `LabelFrame` (líneas 17 a 41). Por lo general, el constructor de la subclase de `JFrame` crea la GUI que se muestra en la ventana, cuando se ejecuta la aplicación. La línea 19 invoca al constructor de la superclase `JFrame` con el argumento "`Prueba de JLabel`". El constructor de `JFrame` utiliza este objeto `String` como el texto en la barra de título de la ventana.

Especificación del esquema

Al crear una GUI, cada componente de ésta debe adjuntarse a un contenedor, como una ventana creada con un objeto `JFrame`. Además, por lo general debemos decidir en dónde *colocar* cada componente de la GUI; esto se conoce como *especificar el esquema*. Como aprenderá al final de este capítulo y en el capítulo 22, Java cuenta con varios **administradores de esquemas** que pueden ayudarle a colocar los componentes.

Muchos entornos de desarrollo integrados (IDE) proporcionan herramientas de diseño de GUI, en las cuales podemos especificar el *tamaño* y la *ubicación* exactos de un componente en forma visual utilizando el ratón; después el IDE genera el código de la GUI por usted. Dichos IDE pueden simplificar en forma considerable la creación de las GUI.

Para asegurar que nuestras GUI puedan utilizarse con *cualquier* IDE, *no* utilizamos un IDE para crear el código de GUI. Usamos administradores de esquemas de Java para *ajustar el tamaño* de los componentes y *posicionarlos*. En el administrador de esquemas `FlowLayout`, los componentes de la GUI se colocan en un *contenedor* de izquierda a derecha, en el orden en el que el programa los une al contenedor. Cuando no hay más espacio para acomodar los componentes en la línea actual, se siguen mostrando de izquierda a derecha en la siguiente línea. Si se *cambia el tamaño* del contenedor, un esquema `FlowLayout` *reordena* los componentes usando menos o más filas con base en la nueva anchura del contenedor. Cada contenedor tiene un *esquema predeterminado*, el cual vamos a cambiar para `LabelFrame` a `FlowLayout` (línea 20). El método `setLayout` se hereda en la clase `LabelFrame`, indirectamente de la clase `Container`. El argumento para el método debe ser un objeto de una clase que implemente la interfaz `LayoutManager` (es decir, `FlowLayout`). La línea 20 crea un nuevo objeto `FlowLayout` y pasa su referencia como argumento para `setLayout`.

Cómo crear y adjuntar `etiqueta1`

Ahora que hemos especificado el esquema de la ventana, podemos empezar a crear y adjuntar componentes de la GUI en la ventana. La línea 23 crea un objeto `JLabel` y pasa "`Etiqueta con texto`" al constructor. El objeto `JLabel` muestra este texto en la pantalla. La línea 24 utiliza el método `setToolTipText` (heredado por `JLabel` de `JComponent`) para especificar la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre el objeto `JLabel` en la GUI. En la segunda captura de pantalla de la figura 12.7 puede ver un cuadro de información sobre herramientas de ejemplo. Cuando ejecute esta aplicación, trate de colocar el ratón sobre cada objeto `JLabel` para ver el cuadro de información sobre herramientas. La línea 25 (figura 12.6) adjunta `etiqueta1` al objeto `LabelFrame`, para lo cual pasa `etiqueta1` al método `add`, que se hereda indirectamente de la clase `Container`.

Error común de programación 12.1

Si no agrega explícitamente un componente de GUI a un contenedor, el componente no se mostrará cuando aparezca el contenedor en la pantalla.

Observación de apariencia visual 12.7

Use cuadros de información sobre herramientas para agregar texto descriptivo a sus componentes de GUI. Este texto ayuda al usuario a determinar el propósito del componente de GUI en la interfaz de usuario.

La interfaz `Icon` y la clase `ImageIcon`

Los iconos son una forma popular de mejorar la apariencia visual de una aplicación, y también se utilizan comúnmente para indicar funcionalidad. Por ejemplo, se utiliza el mismo icono para reproducir la mayoría de los medios de la actualidad en dispositivos como reproductores de DVD y MP3. Varios componentes de Swing pueden mostrar imágenes. Por lo general, un icono se especifica con un argumento **`Icon`** (paquete `javax.swing`) para un constructor o para el método **`setIcon`** del componente. La clase **`ImageIcon`** soporta varios formatos de imágenes, incluyendo el *formato de intercambio de gráficos* (GIF), los *gráficos portables de red* (PNG) y las imágenes del *Grupo de expertos en fotografía unidos* (JPEG).

La línea 28 declara un objeto `ImageIcon`. El archivo `insecto1.png` contiene la imagen a cargar y almacenar en el objeto `ImageIcon`. Esta imagen se incluye en el directorio para este ejemplo. El objeto `ImageIcon` se asigna a la referencia `Icon` llamada `insecto`.

Cómo cargar un recurso de imagen

En la línea 28, la expresión `getClass().getResource("insecto1.png")` invoca al método **`getClass`** (que se hereda de manera indirecta de la clase `Object`) para recuperar una referencia al objeto `Class` que representa la declaración de la clase `LabelFrame`. Después, esa referencia se utiliza para invocar al método **`getResource`** de `Class`, el cual devuelve la ubicación de la imagen como un URL. El constructor de `ImageIcon` utiliza el URL para localizar la imagen, y después la carga en la memoria. Como vimos en el capítulo 1, la JVM carga las declaraciones de las clases en la memoria, usando un cargador de clases. El cargador de clases sabe en qué parte del disco se encuentra localizada cada clase que carga. El método `getResource` utiliza el cargador de clases del objeto `Class` para determinar la *ubicación* de un recurso, como un archivo de imagen. En este ejemplo, el archivo de imagen se almacena en la misma ubicación que el archivo `LabelFrame.class`. Las técnicas aquí descritas permiten que una aplicación cargue archivos de imagen de ubicaciones que son relativas a la ubicación del archivo de clase.

Cómo crear y adjuntar `etiqueta2`

Las líneas 29 y 30 utilizan otro constructor de `JLabel` para crear un objeto `JLabel` que muestre el texto "`Etiqueta con texto e icono`" y el objeto `Icon` llamado `insecto` que se creó en la línea 28. El último argumento del constructor indica que el contenido de la etiqueta está justificado a la izquierda, o con alineación izquierda (es decir, tanto el icono como el texto se encuentran en el lado izquierdo del área de la etiqueta en la pantalla). La interfaz **`SwingConstants`** (paquete `javax.swing`) declara un conjunto de constantes enteras comunes (como `SwingConstants.LEFT`, `SwingConstants.CENTER` y `SwingConstants.RIGHT`) que se utilizan con muchos componentes de Swing. De manera predeterminada, cuando una etiqueta contiene tanto texto como una imagen, el texto aparece a la derecha de una imagen. Las alineaciones horizontal y vertical de un objeto `JLabel` se pueden establecer mediante los métodos **`setHorizontalAlignment`** y **`setVerticalAlignment`**, respectivamente. La línea 31 especifica el texto de información sobre herramientas para `etiqueta2`, y la línea 32 agrega `etiqueta2` al objeto `JFrame`.

Cómo crear y adjuntar `etiqueta3`

La clase `JLabel` cuenta con métodos para modificar la apariencia de una etiqueta, una vez que se crea una instancia de ésta. La línea 34 crea un objeto `JLabel` vacío mediante el constructor sin argumentos. La línea 35 utiliza el método **`setText`** de `JLabel` para establecer el texto mostrado en la etiqueta. El método **`getText`** se puede usar para obtener el texto actual del objeto `JLabel`. La línea 36 utiliza el método `setIcon` de `JLabel` para especificar el objeto `Icon` a mostrar. El método **`getIcon`** se puede usar para obtener el objeto `Icon` actual mostrado en una etiqueta. Las líneas 37 y 38 utilizan los métodos **`setHorizontalTextPosition`** y **`setVerticalTextPosition`** de `JLabel` para especificar la siguiente posición del texto en la etiqueta. En este caso, el texto se centrará en forma *horizontal* y aparecerá en la *parte inferior* de la etiqueta. Por ende, el objeto `Icon` aparecerá *por encima* del texto. Las constantes de posición horizontal en `SwingConstants` son `LEFT`, `CENTER` y `RIGHT` (figura 12.8). Las constantes de

posición vertical en `SwingConstants` son `TOP`, `CENTER` y `BOTTOM` (figura 12.8). La línea 39 (figura 12.6) establece el texto de información sobre [herramientas para `etiqueta3`. La línea 40 agrega `etiqueta3` al objeto `JFrame`.

Constante	Descripción	Constante	Descripción
Constantes de posición horizontal		*Constantes de posición vertical*	
`LEFT`	Coloca el texto a la izquierda	`TOP`	Coloca el texto en la parte superior
`CENTER`	Coloca el texto en el centro	`CENTER`	Coloca el texto en el centro
`RIGHT`	Coloca el texto a la derecha	`BOTTOM`	Coloca el texto en la parte inferior

Fig. 12.8 | Constantes de posicionamiento (miembros `static` de la interfaz `SwingConstants`).

Cómo crear y mostrar una ventana `LabelFrame`

La clase `PruebaLabel` (figura 12.7) crea un objeto de la clase `LabelFrame` (línea 9) y después especifica la operación de cierre predeterminada para la ventana. De manera predeterminada, al cerrar una ventana ésta simplemente se *oculta*. Sin embargo, cuando el usuario cierre la ventana `LabelFrame`, nos gustaría que la aplicación *terminara*. La línea 10 invoca al método `setDefaultCloseOperation` de `LabelFrame` (heredado de la clase `JFrame`) con la constante `JFrame.EXIT_ON_CLOSE` como el argumento para indicar que el programa debe *terminar* cuando el usuario cierre la ventana. Esta línea es importante. Sin ella, la aplicación *no* terminará cuando el usuario cierre la ventana. A continuación, la línea 11 invoca el método `setSize` de `LabelFrame` para especificar la *anchura* y la *altura* de la ventana en *píxeles*. Por último, la línea 12 invoca al método `setVisible` de `LabelFrame` con el argumento `true`, para mostrar la ventana en la pantalla. Pruebe cambiar el tamaño de la ventana, para ver cómo el esquema `FlowLayout` cambia las posiciones de los objetos `JLabel` a medida que la anchura de la ventana cambia.

12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas

Por lo general, un usuario interactúa con la GUI de una aplicación para indicar las tareas que ésta debe realizar. Por ejemplo, cuando usted escribe un mensaje en una aplicación de correo electrónico, al hacer clic en el botón **Enviar** le indica a la aplicación que envíe el correo electrónico a las direcciones especificadas. Las GUI son **controladas por eventos**. Cuando el usuario interactúa con un componente de la GUI, la interacción (conocida como un **evento**) controla el programa para que realice una tarea. Algunas interacciones comunes del usuario que podrían hacer que una aplicación realizara una tarea incluyen el *hacer clic* en un botón, *escribir* en un campo de texto, *seleccionar* un elemento de un menú, *cerrar* una ventana y *mover* el ratón. El código que realiza una tarea en respuesta a un evento se llama **manejador de eventos** y al proceso en general de responder a los eventos se le conoce como **manejo de eventos**.

Vamos a considerar los otros dos componentes de GUI que pueden generar eventos: `JTextField` y `JPasswordField` (paquete `javax.swing`). La clase `JTextField` extiende a la clase `JTextComponent` (paquete `javax.swing.text`), que proporciona muchas características comunes para los componentes de Swing basados en texto. La clase `JPasswordField` extiende a `JTextField` y agrega varios métodos específicos para el procesamiento de contraseñas. Cada uno de estos componentes es un área de una sola línea, en la cual el usuario puede introducir texto mediante el teclado. Las aplicaciones también pueden mostrar texto en un objeto `JTextField` (vea la salida de la figura 12.10). Un objeto `JPasswordField` muestra que se están escribiendo caracteres a medida que el usuario los introduce, pero oculta los caracteres reales con un **carácter de eco**, asumiendo que representan una contraseña que sólo el usuario debe conocer.

Cuando el usuario escribe datos en un objeto `JTextField` o `JPasswordField` y después oprime *Intro*, ocurre un evento. Nuestro siguiente ejemplo demuestra cómo un programa puede realizar una tarea *en respuesta* a ese evento. Las técnicas que se muestran aquí se pueden aplicar a todos los componentes de GUI que generen eventos.

La aplicación de las figuras 12.9 y 12.10 utiliza las clases `JTextField` y `JPasswordField` para crear y manipular cuatro campos de texto. Cuando el usuario escribe en uno de los campos de texto y después oprime *Intro*, la aplicación muestra un cuadro de diálogo de mensaje que contiene el texto que escribió el usuario. Sólo podemos escribir en el campo de texto que esté "**enfocado**". Cuando el usuario *hace clic* en ese componente, éste *recibe el enfoque*. Esto es importante, ya que el campo de texto con el enfoque es el que genera un evento cuando el usuario oprime *Intro*. En este ejemplo, cuando el usuario oprime *Intro* en el objeto `JPasswordField`, se revela la contraseña. Empezaremos por explicar la preparación de la GUI, y después sobre el código para manejar eventos.

```
1  // Fig. 12.9: CampoTextoMarco.java
2  // Los componentes JTextField y JPasswordField.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class CampoTextoMarco extends JFrame
12 {
13    private final JTextField campoTexto1; // campo de texto con tamaño fijo
14    private final JTextField campoTexto2; // campo de texto con texto
15    private final JTextField campoTexto3; // campo de texto con texto y tamaño
16    private final JPasswordField campoContrasenia; // campo de contraseña con texto
17
18    // El constructor de CampoTextoMarco agrega objetos JTextField a JFrame
19    public CampoTextoMarco()
20    {
21       super("Prueba de JTextField y JPasswordField");
22       setLayout(new FlowLayout());
23
24       // construye campo de texto con 10 columnas
25       campoTexto1 = new JTextField(10);
26       add(campoTexto1); // agrega campoTexto1 a JFrame
27
28       // construye campo de texto con texto predeterminado
29       campoTexto2 = new JTextField("Escriba el texto aqui");
30       add(campoTexto2); // agrega campoTexto2 a JFrame
31
32       // construye campo de texto con texto predeterminado y 21 columnas
33       campoTexto3 = new JTextField("Campo de texto no editable", 21);
34       campoTexto3.setEditable(false); // deshabilita la edición
35       add(campoTexto3); // agrega campoTexto3 a JFrame
```

Fig. 12.9 | Objetos `JTextField` y `JPasswordField` (parte 1 de 2).

```
36
37        // construye campo de contraseña con texto predeterminado
38        campoContrasenia = new JPasswordField("Texto oculto");
39        add(campoContrasenia); // agrega campoContrasenia a JFrame
40
41        // registra los manejadores de eventos
42        ManejadorCampoTexto manejador = new ManejadorCampoTexto();
43        campoTexto1.addActionListener(manejador);
44        campoTexto2.addActionListener(manejador);
45        campoTexto3.addActionListener(manejador);
46        campoContrasenia.addActionListener(manejador);
47     }
48
49     // clase interna privada para el manejo de eventos
50     private class ManejadorCampoTexto implements ActionListener
51     {
52        // procesa los eventos de campo de texto
53        @Override
54        public void actionPerformed(ActionEvent evento)
55        {
56           String cadena = "";
57
58           // el usuario oprimió Intro en el objeto JTextField campoTexto1
59           if (evento.getSource() == campoTexto1)
60              cadena = String.format("campoTexto1: %s",
61                 evento.getActionCommand());
62
63           // el usuario oprimió Intro en el objeto JTextField campoTexto2
64           else if (evento.getSource() == campoTexto2)
65              cadena = String.format("campoTexto2: %s",
66                 evento.getActionCommand() );
67
68           // el usuario oprimió Intro en el objeto JTextField campoTexto3
69           else if (evento.getSource() == campoTexto3)
70              cadena = String.format("campoTexto3: %s",
71                 evento.getActionCommand());
72
73           // el usuario oprimió Intro en el objeto JTextField campoContrasenia
74           else if (evento.getSource() == campoContrasenia)
75              cadena = String.format("campoContrasenia: %s",
76                 evento.getActionCommand());
77
78           // muestra el contenido del objeto JTextField
79           JOptionPane.showMessageDialog(null, cadena);
80        }
81     } // fin de la clase interna privada ManejadorCampoTexto
82  } // fin de la clase CampoTextoMarco
```

Fig. 12.9 | Objetos `JTextField` y `JPasswordField` (parte 2 de 2).

La clase `CampoTextoMarco` extiende a `JFrame` y declara tres variables `JTextField` y una variable `JPasswordField` (líneas 13 a 16). Cada uno de los correspondientes campos de texto se instancia y se adjunta al objeto `CampoTextoMarco` en el constructor (líneas 19 a 47).

Creación de la GUI

La línea 22 establece el esquema del objeto `CampoTextoMarco` a `FlowLayout`. La línea 25 crea el objeto `campoTexto1` con 10 columnas de texto. La anchura en *píxeles* de una columna de texto se determina con base en la anchura promedio de un carácter en el tipo de letra actual del campo de texto. Cuando el texto que se muestra es más ancho que el campo de texto en sí, la parte del texto del lado derecho no es visible. Si usted escribe en un campo de texto y el cursor llega al extremo derecho del campo, el texto en el extremo izquierdo se empuja hacia el lado izquierdo del campo y ya no estará visible. Los usuarios pueden usar las flechas de dirección izquierda y derecha para recorrer el texto completo. La línea 26 agrega el objeto `campoTexto1` al objeto `JFrame`.

La línea 29 crea el objeto `campoTexto2` con el texto inicial "`Escriba el texto aqui`" para mostrarlo en el campo de texto. La anchura del campo se determina con base en la anchura del texto predeterminado especificado en el constructor. La línea 30 agrega el objeto `campoTexto2` al objeto `JFrame`.

La línea 33 crea el objeto `campoTexto3` y llama al constructor de `JTextField` con dos argumentos: el texto predeterminado "`Campo de texto no editable`" para mostrarlo y la anchura del campo de texto en columnas (`21`). La línea 34 utiliza el método **`setEditable`** (heredado por `JTextField` de la clase `JTextComponent`) para hacer el campo de texto *no editable*; es decir, el usuario no puede modificar el texto en el campo. La línea 35 agrega el objeto `campoTexto3` al objeto `JFrame`.

La línea 38 crea `campoContrasenia` con el texto "`Texto oculto`" que muestra en el campo de texto. La anchura de este campo de texto se determina con base en la anchura del texto predeterminado. Al ejecutar la aplicación, observe que el texto se muestra como una cadena de asteriscos. La línea 39 agrega `campoContrasenia` al objeto `JFrame`.

Pasos requeridos para establecer el manejo de eventos para un componente de GUI

Este ejemplo debe mostrar un diálogo de mensaje que contenga el texto de un campo de texto, cuando el usuario oprime *Intro* en ese campo. Antes de que una aplicación pueda responder a un evento para un componente de GUI específico, debemos:

1. Crear una clase que represente al manejador de eventos e implemente una interfaz apropiada, conocida como **interfaz de escucha de eventos**.
2. Indicar que se debe notificar a un objeto de la clase del *paso 1* cuando ocurra el evento. A esto se le conoce como **registrar el manejador de eventos**.

Uso de una clase anidada para implementar un manejador de eventos

Todas las clases que hemos visto hasta ahora se conocen como **clases de nivel superior**; es decir, las clases no se declararon *dentro* de otra clase. Java nos permite declarar clases *dentro* de otras clases; a éstas se les conoce como **clases anidadas**. Las clases anidadas pueden ser `static` o no `static`. Las clases anidadas no `static` se llaman **clases internas**, y se utilizan con frecuencia para implementar *manejadores de eventos*.

Un objeto de una clase interna debe crearse mediante un objeto de la clase de nivel superior que contenga a la clase interna. Cada objeto de la clase interna tiene *implícitamente* una referencia a un objeto de su clase de nivel superior. El objeto de la clase interna puede usar esta referencia implícita para acceder directamente a todas las variables y métodos de la clase de nivel superior. Una clase interna que es `static` no requiere un objeto de su clase de nivel superior, y no tiene una referencia implícita a un objeto de la clase de nivel superior. Como veremos en el capítulo 13, Gráficos y Java 2D, la API de gráficos 2D de Java utiliza mucho las clases anidadas `static`.

La clase interna `ManejadorCampoTexto`

El manejo de eventos en este ejemplo se realiza mediante un objeto de la clase interna `private` `ManejadorCampoTexto` (líneas 50 a 81). Esta clase es `private` debido a que se utilizará sólo para crear manejadores de eventos para los campos de texto en la clase de nivel superior `CampoTextoMarco`. Al igual que con los otros miembros de una clase, las *clases internas* pueden declararse como `public`, `protected` o

`private`. Puesto que los manejadores de eventos tienden a ser específicos para la aplicación en la que están definidos, a menudo se implementan como clases internas `private` o como *clases internas anónimas* (sección 12.11).

Los componentes de GUI pueden generar una variedad de eventos en respuesta a las interacciones del usuario. Cada evento se representa mediante una clase, y sólo puede procesarse mediante el tipo apropiado de manejador de eventos. En la mayoría de los casos, los eventos que soporta un componente se describen en la documentación de la API de Java para la clase de ese componente y sus superclases. Cuando el usuario oprime *Intro* en un objeto `JTextField` o `JPasswordField`, ocurre un evento **`ActionEvent`** (paquete `java.awt.event`). Dicho evento se procesa mediante un objeto que implementa la interfaz **`ActionListener`** (paquete `java.awt.event`). La información aquí descrita está disponible en la documentación de la API de Java para las clases `JTextField` y `ActionEvent`. Como `JPasswordField` es una subclase de `JTextField`, `JPasswordField` soporta los mismos eventos.

Para prepararnos para manejar los eventos en este ejemplo, la clase interna `ManejadorCampoTexto` implementa la interfaz `ActionListener` y declara el único método en esa interfaz: `actionPerformed` (líneas 53 a 80). Este método especifica las tareas a realizar cuando ocurre un evento `ActionEvent`. Por lo tanto, la clase `ManejadorCampoTexto` cumple con el *paso 1* que se listó anteriormente en esta sección. En breve hablaremos sobre los detalles del método `actionPerformed`.

Registro del manejador de evento para cada campo de texto

En el constructor de `CampoTextoMarco`, la línea 42 crea un objeto `ManejadorCampoTexto` y lo asigna a la variable `manejador`. El método `actionPerformed` de este objeto se llamará en forma automática cuando el usuario oprima *Intro* en cualquiera de los campos de texto de la GUI. Sin embargo, antes de que pueda ocurrir esto, el programa debe registrar este objeto como el manejador de eventos para cada campo de texto. Las líneas 43 a 46 son las instrucciones de *registro de eventos* que especifican a `manejador` como el manejador de eventos para los tres objetos `JTextField` y el objeto `JPasswordField`. La aplicación llama al método **`addActionListener`** de `JTextField` para registrar el manejador de eventos para cada componente. Este método recibe como argumento un objeto `ActionListener`, el cual puede ser un objeto de cualquier clase que implemente a `ActionListener`. El objeto `manejador` *es un* `ActionListener`, ya que la clase `ManejadorCampoTexto` implementa a `ActionListener`. Una vez que se ejecutan las líneas 43 a 46, el objeto `manejador` **escucha los eventos**. Ahora, cuando el usuario oprime *Intro* en cualquiera de estos cuatro campos de texto, se hace una llamada al método `actionPerformed` (líneas 53 a 80) en la clase `ManejadorCampoTexto` para que maneje el evento. Si *no* está registrado un manejador de eventos para un campo de texto específico, el evento que ocurre cuando el usuario oprime *Intro* en ese campo se **consume**; es decir, la aplicación simplemente lo *ignora*.

Observación de ingeniería de software 12.1

El componente de escucha de eventos para cierto evento debe implementar a la interfaz de escucha de eventos apropiada.

Error común de programación 12.2

Si olvida registrar un objeto manejador de eventos para un tipo de evento específico de un componente de la GUI, los eventos de ese tipo serán ignorados.

Detalles del método `actionPerformed` *de la clase* `ManejadorCampoTexto`

En este ejemplo estamos usando el método `actionPerformed` de un objeto manejador de eventos (líneas 53 a 80) para manejar los eventos generados por cuatro campos de texto. Como nos gustaría imprimir en pantalla el nombre de la variable de instancia de cada campo de texto para fines demostrativos, debemos

determinar *cuál* campo de texto generó el evento cada vez que se hace una llamada a `actionPerformed`. El componente con el que interactúa el usuario es el **origen del evento**. Cuando el usuario oprime *Intro* mientras uno de los campos de texto o el campo contraseña *tiene el enfoque*, el sistema crea un objeto `ActionEvent` único que contiene información acerca del evento que acaba de ocurrir, como el origen del evento y el texto en el campo de texto. Después, el sistema pasa este objeto `ActionEvent` en una llamada al método `actionPerformed` del componente de escucha de eventos. La línea 56 declara el objeto `String` que se va a mostrar. La variable se inicializa con la **cadena vacía**: un objeto `String` que no contiene caracteres. En caso de que no se ejecute ninguna de las bifurcaciones de la instrucción `if` anidada en las líneas 59 a 76, el compilador requiere que se inicialice la variable.

El método `getSource` de `ActionEvent` (que se llama en las líneas 59, 64, 69 y 74) devuelve una referencia al origen del evento. La condición en la línea 59 pregunta, "¿Es `campoTexto1` el origen del evento?" Esta condición compara las referencias en ambos lados del operador `==` para determinar si se refieren al mismo objeto. Si *ambos* se refieren a `campoTexto1`, el usuario oprimió *Intro* en `campoTexto1`. Después, las líneas 60 y 61 crean un objeto `String` que contiene el mensaje que la línea 79 mostrará en un diálogo de mensaje. La línea 61 utiliza el método **`getActionCommand`** de `ActionEvent` para obtener el texto que escribió el usuario en el campo de texto que generó el evento.

En este ejemplo, mostramos el texto de la contraseña en el objeto `JPasswordField` cuando el usuario oprime *Intro* en ese campo. Algunas veces es necesario procesar mediante programación los caracteres en una contraseña. El método **`getPassword`** de la clase `JPasswordField` devuelve los caracteres de la contraseña como un arreglo de tipo `char`.

La clase `PruebaCampoTexto`

La clase `PruebaCampoTexto` (figura 12.10) contiene el método `main` que ejecuta esta aplicación y muestra un objeto de la clase `CampoTextoMarco`. Al ejecutar la aplicación, incluso el campo `JTextField` (`campoTexto3`), que no se puede editar, puede generar un evento `ActionEvent`. Para probar esto, haga clic en el campo de texto para darle el enfoque y después oprima *Intro*. Además, el texto actual de la contraseña se muestra al oprimir *Intro* en el campo `JPasswordField`. ¡Desde luego que generalmente no se debe mostrar la contraseña!

```
1  // Fig. 12.10: PruebaCampoTexto.java
2  // Prueba de CampoTextoMarco.
3  import javax.swing.JFrame;
4
5  public class PruebaCampoTexto
6  {
7     public static void main(String[] args)
8     {
9        CampoTextoMarco campoTextoMarco = new CampoTextoMarco();
10       campoTextoMarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       campoTextoMarco.setSize(350, 100);
12       campoTextoMarco.setVisible(true);
13    }
14 } // fin de la clase PruebaCampoTexto
```

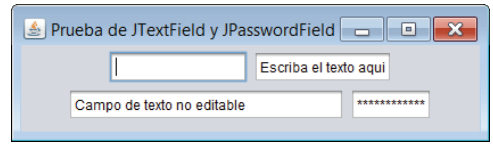


Fig. 12.10 | Prueba de `CampoTextoMarco` (parte 1 de 2).

Fig. 12.10 | Prueba de `CampoTextoMarco` (parte 2 de 2).

Esta aplicación usó un solo objeto de la clase `ManejadorCampoTexto` como el componente de escucha de eventos para cuatro campos de texto. Empezando en la sección 12.10, verá que es posible declarar varios objetos de escucha de eventos del mismo tipo, y registrar cada objeto para cada evento de un componente de la GUI por separado. Esta técnica nos permite eliminar la lógica `if...else` utilizada en el manejador de eventos de este ejemplo, al proporcionar manejadores de eventos separados para los eventos de cada componente.

Java SE 8: cómo implementar componentes de escucha de eventos con lambdas

Recuerde que las interfaces como `ActionListener` que tienen sólo un método `abstract` son interfaces funcionales en Java SE 8. En la sección 17.9 le mostraremos una forma más concisa de implementar dichas interfaces de escucha de eventos con las lambdas de Java SE 8.

12.7 Tipos de eventos comunes de la GUI e interfaces de escucha

En la sección 12.6 aprendió que la información acerca del evento que ocurre cuando el usuario oprime *Intro* en un campo de texto se almacena en un objeto `ActionEvent`. Pueden ocurrir muchos tipos distintos de eventos cuando el usuario interactúa con una GUI. La información acerca de cualquier evento se almacena en un objeto de una clase que extiende a `AWTEvent` (del paquete `java.awt`). La figura 12.11 ilustra una jerarquía que contiene muchas clases de eventos del paquete **`java.awt.event`**. Algunas de éstas se describen en este capítulo y en el capítulo 22. Estos tipos de eventos se utilizan tanto con componentes

de AWT como de Swing. Los tipos de eventos adicionales que son específicos para los componentes de GUI de Swing se declaran en el paquete `javax.swing.event`.

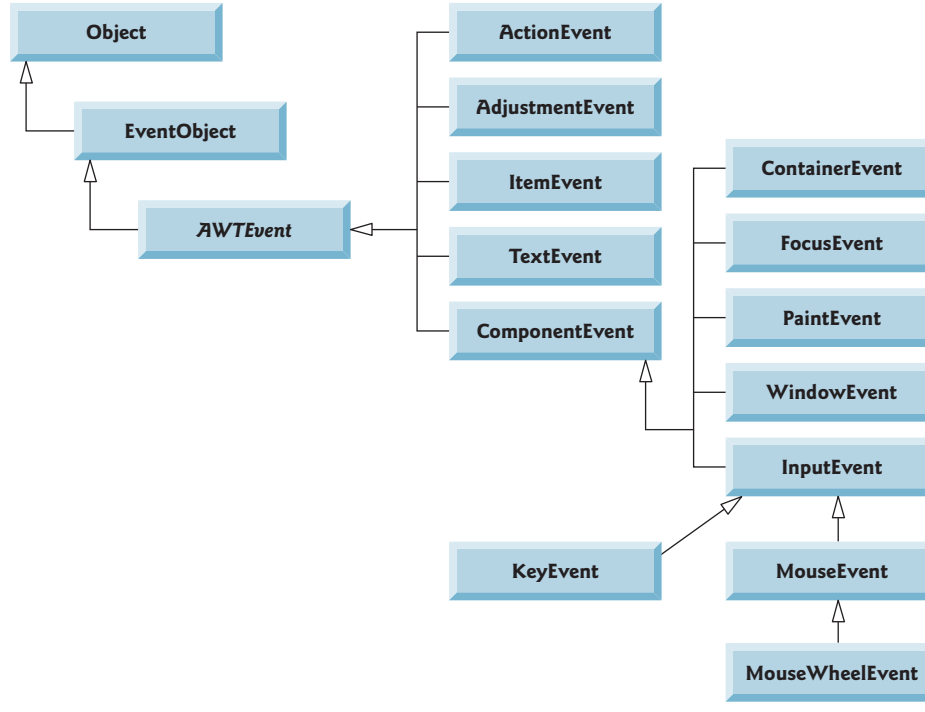


Fig. 12.11 | Algunas clases de eventos del paquete `java.awt.event`.

Resumiremos las tres partes requeridas para el mecanismo de manejo de eventos que vimos en la sección 12.6: el *origen del evento*, el *objeto del evento* y el *componente de escucha del evento*. El origen del evento es el componente específico de la GUI con el que interactúa el usuario. El objeto del evento encapsula información acerca del evento que ocurrió, como una referencia al origen del mismo, y cualquier información específica del evento que pueda requerir el componente de escucha del evento, para que pueda manejarlo. El componente de escucha del evento es un objeto que recibe una notificación del origen del evento cuando éste ocurre; en efecto, "escucha" un evento, y uno de sus métodos se ejecuta en respuesta al evento. Un método del componente de escucha del evento recibe un objeto evento cuando se notifica al componente de escucha acerca del evento. Después, el componente de escucha del evento utiliza el objeto evento para responder. A este modelo de manejo de eventos se le conoce como **modelo de eventos por delegación**, ya que el procesamiento de un evento se delega a un objeto específico (el componente de escucha de eventos) de la aplicación.

Para cada tipo de objeto evento, hay por lo general una interfaz de escucha de eventos que le corresponde. Un componente de escucha de eventos para un evento de GUI es un objeto de una clase que implementa a una o más de las interfaces de escucha de eventos de los paquetes `java.awt.event` y `javax.swing.event`. Muchos de los tipos de componentes de escucha de eventos son comunes para los componentes de Swing y de AWT. Dichos tipos se declaran en el paquete `java.awt.event`, y algunos de ellos se muestran en la figura 12.12. Los tipos de escucha de eventos adicionales específicos para los componentes de Swing se declaran en el paquete `javax.swing.event`.

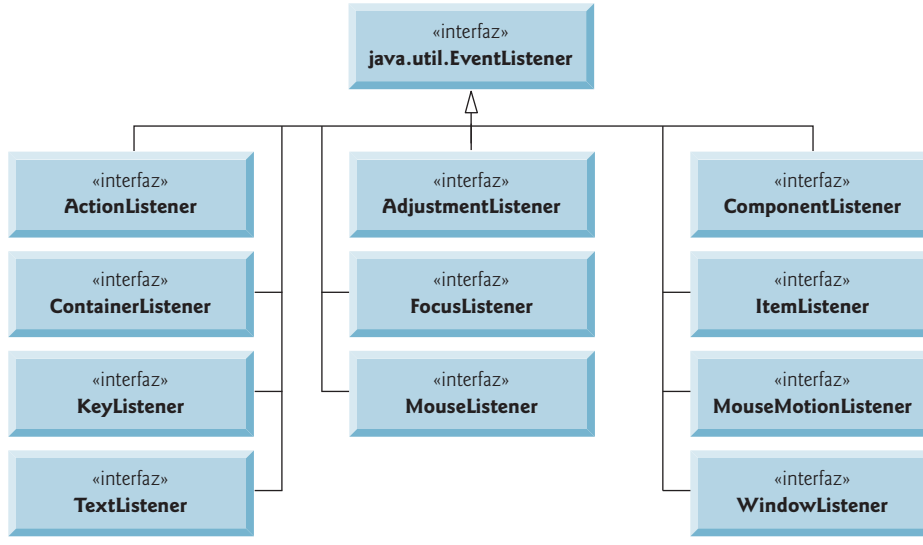


Fig. 12.12 | Algunas interfaces comunes de componentes de escucha de eventos del paquete `java.awt.event`.

Cada interfaz de escucha de eventos especifica uno o más métodos manejadores de eventos que *deben* declararse en la clase que implementa a la interfaz. En la sección 10.9 vimos que cualquier clase que implementa a una interfaz debe declarar a *todos* los métodos `abstract` de esa interfaz; en caso contrario, la clase es `abstract` y no puede utilizarse para crear objetos.

Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus *componentes de escucha registrados*, llamando al *método de manejo de eventos* apropiado de cada componente de escucha. Por ejemplo, cuando el usuario oprime la tecla *Intro* en un objeto `JTextField`, se hace una llamada al método `actionPerformed` del componente de escucha registrado. En la siguiente sección completaremos nuestro análisis sobre cómo funciona el manejo de eventos del ejemplo anterior.

12.8 Cómo funciona el manejo de eventos

Vamos a ilustrar cómo funciona el mecanismo de manejo de eventos, utilizando a `campoTexto1` del ejemplo de la figura 12.9. Tenemos dos preguntas sin contestar de la sección 12.7:

1. ¿Cómo se *registró* el *manejador de eventos*?
2. ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a algún otro método manejador de eventos?

La primera pregunta se responde mediante el registro de eventos que se lleva a cabo en las líneas 43 a 46 de la figura 12.9. En la figura 12.13 se muestra un diagrama de la variable `JTextField` llamada `campoTexto1`, de la variable `ManejadorCampoTexto` llamada `manejador` y de los objetos a los que hacen referencia.

Registro de eventos

Todo `JComponent` tiene una variable de instancia llamada `listenerList`, que hace referencia a un objeto de la clase `EventListenerList` (paquete `javax.swing.event`). Cada objeto de una subclase de `JComponent` mantiene referencias a todos sus *componentes de escucha registrados* en `listenerList`. Por simplicidad, hemos colocado a `listenerList` en el diagrama como un arreglo, abajo del objeto `JTextField` en la figura 12.13.

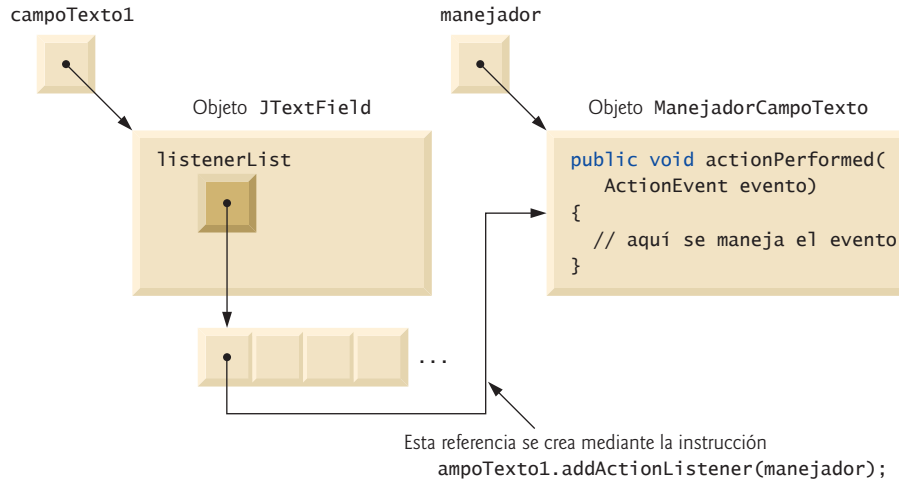


Fig. 12.13 | Registro de eventos para el objeto `JTextField` `campoTexto1`.

Cuando se ejecuta la siguiente instrucción (línea 43 de la figura 12.9):

```
campoTexto1.addActionListener(manejador);
```

se coloca en el objeto `listenerList` de `campoTexto1` una nueva entrada que contiene una referencia al objeto `ManejadorCampoTexto`. Aunque no se muestra en el diagrama, esta nueva entrada también incluye el tipo del componente de escucha (`ActionListener`). Mediante el uso de este mecanismo, cada componente ligero de GUI de Swing mantiene su propia lista de *componentes de escucha* que se *registraron* para *manejar* los *eventos* del componente.

Invocación al manejador de eventos

El tipo de componente de escucha de eventos es importante para responder a la segunda pregunta: ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a otro método? Todo componente de la GUI soporta varios *tipos de eventos*, incluyendo **eventos de ratón**, **eventos de tecla** y otros más. Cuando ocurre un evento, éste se **despacha** solamente a los *componentes de escucha de eventos* del tipo apropiado. El despachamiento (*dispatching*) es simplemente el proceso por el cual el componente de la GUI llama a un método manejador de eventos en cada uno de sus componentes de escucha registrados para el tipo de evento que ocurrió.

Cada *tipo de evento* tiene una o más *interfaces de escucha de eventos* correspondientes. Por ejemplo, los eventos tipo `ActionEvent` son manejados por objetos `ActionListener`, los eventos tipo `MouseEvent` son manejados por objetos `MouseListener` y `MouseMotionListener`, y los eventos tipo `KeyEvent` son manejados por objetos `KeyListener`. Cuando ocurre un evento, el componente de la GUI recibe (de la JVM) un ***ID de evento*** único, el cual especifica el tipo de evento. El componente de la GUI utiliza el ID de evento para decidir a cuál tipo de componente de escucha debe despacharse el evento, y para decidir cuál método llamar en cada objeto de escucha. Para un `ActionEvent`, el evento se despacha al método `actionPerformed` de *todos* los objetos `ActionListener` registrados (el único método en la interfaz `ActionListener`). En el caso de un `MouseEvent`, el evento se despacha a *todos* los objetos `MouseListener` o `MouseMotionListener` registrados, dependiendo del evento de ratón que ocurra. El ID de evento del objeto `MouseListener` determina cuáles de los diversos métodos manejadores de eventos de ratón son llamados. Todas estas decisiones

las administran los componentes de la GUI por usted. Todo lo que usted necesita hacer es registrar un manejador de eventos para el tipo de evento específico que requiere su aplicación, y el componente de GUI asegurará que se llame al método apropiado del manejador de eventos cuando ocurra el evento. Hablaremos sobre otros tipos de eventos e interfaces de escucha de eventos a medida que se vayan necesitando, con cada nuevo componente que vayamos introduciendo.

Tip de desempeño 12.1

Las GUI siempre deben seguir respondiendo al usuario. Si se realiza una tarea que tarde mucho tiempo en ejecutarse en un manejador de eventos, el usuario no podrá interactuar con la GUI sino hasta que se complete la tarea. La sección 23.11 demuestra técnicas para evitar dichos problemas.

12.9 JButton

Un **botón** es un componente en el que el usuario hace clic para desencadenar cierta acción. Una aplicación de Java puede utilizar varios tipos de botones, incluyendo **botones de comando**, **casillas de verificación**, **botones interruptores** y **botones de opción**. En la figura 12.14 se muestra la jerarquía de herencia de los botones de Swing que veremos en este capítulo. Como puede ver en el diagrama, todos los tipos de botones son subclases de `AbstractButton` (paquete `javax.swing`), la cual declara las características comunes para los botones de Swing. En esta sección nos concentraremos en los botones que se utilizan comúnmente para iniciar un comando.

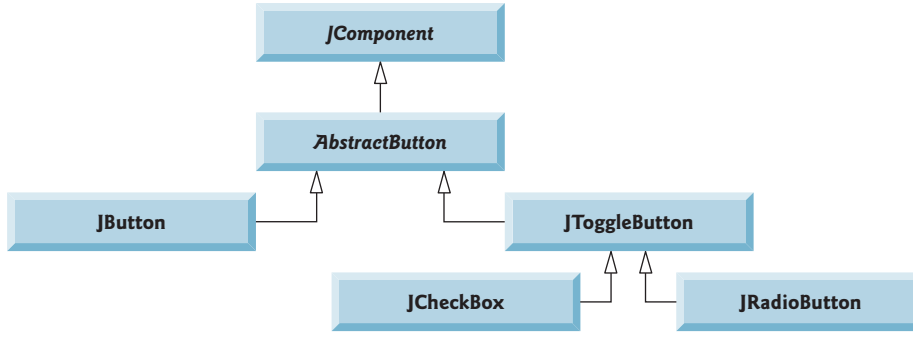


Fig. 12.14 | Jerarquía de botones de Swing.

Un *botón de comando* (vea la salida de la figura 12.16) genera un evento `ActionEvent` cuando el usuario hace clic en él. Los botones de comando se crean con la clase `JButton`. El texto en la cara de un objeto `JButton` se llama **etiqueta del botón**.

Observación de apariencia visual 12.8

El texto en los botones por lo general usa mayúsculas para las primeras letras de las palabras relevantes.

Observación de apariencia visual 12.9

Una GUI puede tener muchos objetos `JButton`, pero cada etiqueta de botón debe ser única en las partes de la GUI en que se muestre. Tener más de un `JButton` con la misma etiqueta hace que los objetos `JButton` sean ambiguos para el usuario.

La aplicación de las figuras 12.15 y 12.16 crea dos objetos `JButton` y demuestra que estos objetos tienen soporte para mostrar objetos `Icon`. El manejo de eventos para los botones se lleva a cabo mediante una sola instancia de la *clase interna* `ManejadorBoton` (figura 12.15, líneas 39 a 48).

```
1  // Fig. 12.15: MarcoBoton.java
2  // Botones de comando y eventos de acción.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8  import javax.swing.Icon;
9  import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class MarcoBoton extends JFrame
13 {
14    private final JButton botonJButtonSimple; // botón con texto solamente
15    private final JButton botonJButtonElegante; // botón con iconos
16
17    // MarcoBoton agrega objetos JButton a JFrame
18    public MarcoBoton()
19    {
20       super("Prueba de botones");
21       setLayout(new FlowLayout());
22
23       botonJButtonSimple = new JButton("Boton simple"); // botón con texto
24       add(botonJButtonSimple); // agrega botonJButtonSimple a JFrame
25
26       Icon insecto1 = new ImageIcon(getClass().getResource("insecto1.gif"));
27       Icon insecto2 = new ImageIcon(getClass().getResource("insecto2.gif"));
28       botonJButtonElegante = new JButton("Boton elegante", insecto1); // establece
                                                                            la imagen
29       botonJButtonElegante.setRolloverIcon(insecto2); // establece la imagen de
                                                           sustitución
30       add(botonJButtonElegante); // agrega botonJButtonElegante a JFrame
31
32       // crea nuevo ManejadorBoton para manejar los eventos de botón
33       ManejadorBoton manejador = new ManejadorBoton();
34       botonJButtonElegante.addActionListener(manejador);
35       botonJButtonSimple.addActionListener(manejador);
36    }
37
38    // clase interna para manejar eventos de botón
39    private class ManejadorBoton implements ActionListener
40    {
41       // maneja evento de botón
42       @Override
43       public void actionPerformed(ActionEvent evento)
44       {
45          JOptionPane.showMessageDialog(MarcoBoton.this, String.format(
46             "Usted oprimio: %s", evento.getActionCommand()));
47       }
48    }
49 } // fin de la clase MarcoBoton
```

Fig. 12.15 | Botones de comando y eventos de acción.

```
1  // Fig. 12.16: PruebaBoton.java
2  // Prueba de MarcoBoton.
3  import javax.swing.JFrame;
4
5  public class PruebaBoton
6  {
7     public static void main(String[] args)
8     {
9        MarcoBoton marcoBoton = new MarcoBoton();
10       marcoBoton.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoBoton.setSize(275, 110);
12       marcoBoton.setVisible(true);
13    }
14 } // fin de la clase PruebaBoton
```

Fig. 12.16 | Prueba de `MarcoBoton`.

En las líneas 14 y 15 se declaran las variables `botonJButtonSimple` y `botonJButtonElegante` de la clase `JButton`. Los correspondientes objetos se instancian en el constructor. En la línea 23 se crea `botonJButtonSimple` con la etiqueta "`Boton simple`". En la línea 24 se agrega el botón al objeto `JFrame`.

Un objeto `JButton` puede mostrar un objeto `Icon`. Para proveer al usuario un nivel adicional de interacción visual con la GUI, un objeto `JButton` puede tener también un objeto **Icon de sustitución**, que es un `Icon` que se muestre cuando el usuario coloque el ratón encima del `JButton`. El icono en el botón `JButton` cambia a medida que el ratón se mueve hacia dentro y fuera del área del botón en la pantalla. En las líneas 26 y 27 se crean dos objetos `ImageIcon` que representan al objeto `Icon` predeterminado y el objeto `Icon` de sustitución para el objeto `JButton` creado en la línea 28. Ambas instrucciones

suponen que los archivos de imagen están guardados en el *mismo* directorio que la aplicación. Por lo general, las imágenes se colocan en el *mismo* directorio que la aplicación o en un subdirectorio como `images`). Estos archivos de imágenes se incluyen en el ejemplo.

En la línea 28 se crea `botonJButtonElegante` con el texto "`Boton elegante`" y el icono `insecto1`. De manera predeterminada, el texto se muestra a la *derecha* del icono. En la línea 29 se utiliza el método **`setRolloverIcon`** (heredado de la clase `AbstractButton`) para especificar la imagen a mostrar en el objeto `JButton` cuando el usuario coloque el ratón sobre el botón. En la línea 30 se agrega el `JButton` al objeto `JFrame`.

Observación de apariencia visual 12.10

Debido a que la clase `AbstractButton` soporta texto e imágenes en un botón, todas las subclases de `AbstractButton` soportan también texto e imágenes.

Observación de apariencia visual 12.11

Los iconos de sustitución proveen una retroalimentación visual que les indica que ocurrirá una acción al hacer clic en un botón `JButton`.

Los objetos `JButton`, al igual que los objetos `JTextField`, generan eventos `ActionEvent` que pueden ser procesados por cualquier objeto `ActionListener`. En las líneas 33 a 35 se crea un objeto de la *clase interna* `private` `ManejadorBoton` y se usa `addActionListener` para *registrarlo* como el *manejador de eventos* para cada objeto `JButton`. La clase `ManejadorBoton` (líneas 39 a 48) declara a `actionPerformed` para mostrar un cuadro de diálogo de mensaje que contiene la etiqueta del botón que el usuario oprimió. Para un evento de `JButton`, el método `getActionCommand` de `ActionEvent` devuelve la etiqueta del objeto `JButton`.

Cómo acceder a la referencia `this` en un objeto de una clase de nivel superior desde una clase interna

Cuando ejecute esta aplicación y haga clic en uno de sus botones, observe que el diálogo de mensaje que aparece está centrado sobre la ventana de la aplicación. Esto ocurre debido a que la llamada al método `showMessageDialog` de `JOptionPane` (líneas 44 y 45) utiliza a `MarcoBoton.this`, en vez de `null` como el primer argumento. Cuando este argumento no es `null`, representa lo que se denomina el *componente de GUI padre* del diálogo de mensaje (en este caso, la ventana de aplicación es el componente padre) y permite centrar el diálogo sobre ese componente, cuando se muestra el diálogo. `MarcoBoton.this` representa a la referencia `this` del objeto de la clase `MarcoBoton` de nivel superior.

Observación de ingeniería de software 12.2

Cuando se utiliza en una clase interna, la palabra clave `this` se refiere al objeto actual de la clase interna que se está manipulando. Un método de la clase interna puede utilizar la referencia `this` del objeto de su clase externa, si antepone a `this` el nombre de la clase externa y un punto, como en **`MarcoBoton.this`**.

12.10 Botones que mantienen el estado

Los componentes de la GUI de Swing contienen tres tipos de **botones de estado** (**`JToggleButton`**, **`JCheckBox`** y **`JRadioButton`**), los cuales tienen valores encendido/apagado o verdadero/falso. Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton` (figura 12.14). Un objeto `JRadioButton` es distinto de un objeto `JCheckBox` en cuanto a que por lo general hay varios objetos `JRadioButton` que se agrupan y son *mutuamente excluyentes*; es decir, sólo *uno* de los objetos en el grupo puede estar seleccionado en un momento dado, de igual forma que los botones en la radio de un auto. Primero veremos la clase `JCheckBox`.

12.10.1 JCheckBox

La aplicación de las figuras 12.17 y 12.18 utilizan dos objetos `JCheckBox` para seleccionar el estilo deseado de letra para el texto a mostrar en un objeto `JTextField`. Cuando se selecciona, uno aplica un estilo en negrita y el otro aplica un estilo en cursivas. Si *ambos* se seleccionan, el estilo del tipo de letra es negrita *y* cursiva. Cuando la aplicación se ejecuta por primera vez, ninguno de los objetos `JCheckBox` está activado (es decir, ambos son `false`), por lo que el tipo de letra es de texto plano. La clase `PruebaCheckBox` (figura 12.18) contiene el método `main` que ejecuta esta aplicación.

```
1  // Fig. 12.17: MarcoCasillaVerificacion.java
2  // Botones JcheckBox y eventos de elementos.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JCheckBox;
10
11 public class MarcoCasillaVerificacion extends JFrame
12 {
13    private JTextField campoTexto; // muestra el texto en tipos de letra
                                     cambiantes
14    private JCheckBox negritaJCheckBox; // para seleccionar/deseleccionar negrita
15    private JCheckBox cursivaJCheckBox; // para seleccionar/deseleccionar cursiva
16
17    // El constructor de MarcoCasillaVerificacion agrega objetos JCheckBox a JFrame
18    public MarcoCasillaVerificacion()
19    {
20       super("Prueba de JCheckBox");
21       setLayout(new FlowLayout());
22
23       // establece JTextField y su tipo de letra
24       campoTexto = new JTextField("Observe como cambia el estilo de tipo de
                                      letra", 20);
25       campoTexto.setFont(new Font("Serif", Font.PLAIN, 14));
26       add(campoTexto); // agrega campoTexto a JFrame
27
28       negritaJCheckBox = new JCheckBox("Negrita");
29       cursivaJCheckBox = new JCheckBox("Cursiva");
30       add(negritaJCheckBox); // agrega casilla de verificación "negrita" a JFrame
31       add(cursivaJCheckBox); // agrega casilla de verificación "cursiva" a JFrame
32
33       // registra componentes de escucha para objetos JCheckBox
34       ManejadorCheckBox manejador = new ManejadorCheckBox();
35       negritaJCheckBox.addItemListener(manejador);
36       cursivaJCheckBox.addItemListener(manejador);
37    }
38
39    // clase interna privada para el manejo de eventos ItemListener
40    private class ManejadorCheckBox implements ItemListener
41    {
```

Fig. 12.17 | Botones `JCheckBox` y eventos de los elementos (parte 1 de 2).

```
42        // responde a los eventos de casilla de verificación
43        @Override
44        public void itemStateChanged(ItemEvent evento)
45        {
46           Font tipoletra = null; // almacena el nuevo objeto Font
47
48           // determina cuáles objetos CheckBox están seleccionados
                y crea el objeto Font
49           if (negritaJCheckBox.isSelected() && cursivaJCheckBox.isSelected())
50              tipoletra = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
51           else if (negritaJCheckBox.isSelected())
52              tipoletra = new Font("Serif", Font.BOLD, 14);
53           else if (cursivaJCheckBox.isSelected())
54              tipoletra = new Font("Serif", Font.ITALIC, 14);
55           else
56              tipoletra = new Font("Serif", Font.PLAIN, 14);
57
58           campoTexto.setFont(tipoletra);
59        }
60     }
61  } // fin de la clase MarcoCasillaVerificacion
```

Fig. 12.17 | Botones `JCheckBox` y eventos de los elementos (parte 2 de 2).

```
1  // Fig. 12.18: PruebaCasillaVerificacion.java
2  // Prueba de MarcoCasillaVerificacion.
3  import javax.swing.JFrame;
4
5  public class PruebaCasillaVerificacion
6  {
7     public static void main(String[] args)
8     {
9        MarcoCasillaVerificacion marcoCasillaVerificacion = new MarcoCasillaVerifi-
                                                             cacion();
10       marcoCasillaVerificacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoCasillaVerificacion.setSize(275, 100);
12       marcoCasillaVerificacion.setVisible(true);
13    }
14 } // fin de la clase PruebaCasillaVerificacion
```

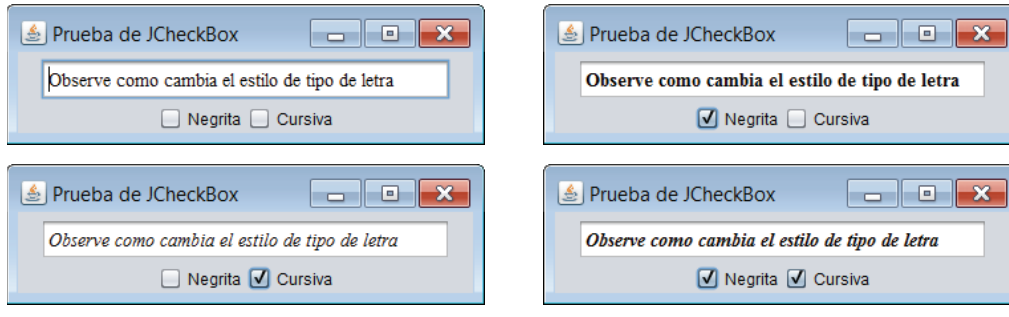


Fig. 12.18 | Prueba de `MarcoCasillaVerificacion`.

Una vez creado e inicializado el objeto `JTextField` (figura 12.17, línea 24), en la línea 25 se utiliza el método **`setFont`** (heredado por `JTextField` indirectamente de la clase `Component`) para establecer el tipo de letra del objeto `JTextField` con un nuevo objeto de la clase **`Font`** (paquete `java.awt`). El nuevo objeto `Font` se inicializa con "`Serif`" (un nombre de tipo de letra genérico que parecido a Times y se soporta en todas las plataformas de Java), estilo `Font.PLAIN` y tamaño de 14 puntos. A continuación, en las líneas 28 y 29 se crean dos objetos `JCheckBox`. El objeto `String` que se pasa al constructor de `JCheckBox` es la **etiqueta de la casilla de verificación** que aparece de manera predeterminada a la derecha del objeto `JCheckBox`.

Cuando el usuario hace clic en un objeto `JCheckBox`, ocurre un evento **`ItemEvent`**. Este evento puede manejarse mediante un objeto **`ItemListener`**, que *debe* implementar al método **`itemStateChanged`**. En este ejemplo, el manejo de eventos se lleva a cabo mediante una instancia de la *clase interna* `private ManejadorCheckBox` (o `ManejadorCasillaVerificacion`) (líneas 40 a 60). En las líneas 34 a 36 se crea una instancia de la clase `ManejadorCheckBox` y se registra con el método **`addItemListener`** como componente de escucha para ambos objetos `JCheckBox`.

El método `itemStateChanged` (líneas 43 a 59) de `ManejadorCheckBox` es llamado cuando el usuario hace clic en el objeto `negritaJCheckBox` o `cursivaJCheckBox`. En este ejemplo, no necesitamos saber en cuál de los dos objetos `JCheckBox` se hizo clic, ya que usamos ambos estados para determinar el tipo de letra a mostrar. En la línea 49 se utiliza el método **`isSelected`** de `JCheckBox` para determinar si ambos objetos `JCheckBox` están seleccionados. De ser así, la línea 50 crea un tipo de letra en negrita y cursiva, sumando las constantes `Font.BOLD` y `Font.ITALIC` para el argumento de estilo de letra del constructor de `Font`. La línea 51 determina si la casilla `negritaJCheckBox` está seleccionada, y de ser así en la línea 52 se crea un tipo de letra en negrita. La línea 53 determina si está seleccionada la casilla `cursivaJCheckBox`, y de ser así en la línea 54 se crea un tipo de letra en cursiva. Si ninguna de las condiciones anteriores es verdadera, en la línea 56 se crea un tipo de letra simple usando la constante `Font.PLAIN` de `Font`. Por último, en la línea 58 se establece el nuevo tipo de letra de `campoTexto`, el cual cambia el tipo de letra en el objeto `JTextField` en pantalla.

Relación entre una clase interna y su clase de nivel superior

La clase `ManejadorCheckBox` utilizó las variables `negritaJCheckBox` (líneas 49 y 51), `cursivaJCheckBox` (líneas 49 y 53) y `campoTexto` (línea 58), aun cuando estas variables *no* se declaran en la clase interna. Recuerde que una *clase interna* tiene una relación especial con su *clase de nivel superior*, por lo que se le permite acceder a *todas* las variables y métodos de la clase de nivel superior. El método `itemStateChanged` (líneas 43 a 59) de la clase `ManejadorCheckBox` utiliza esta relación para determinar cuáles objetos `JCheckBox` están seleccionados y para establecer el tipo de letra en el objeto `JTextField`. Observe que ninguna parte del código en la clase interna `ManejadorCheckBox` requiere una referencia explícita al objeto de la clase de nivel superior.

12.10.2 `JRadioButton`

Los **botones de opción** (que se declaran con la clase `JRadioButton`) son similares a las casillas de verificación, en cuanto a que tienen dos estados: *seleccionado* y *no seleccionado* (al que también se le conoce como *deseleccionado*). Sin embargo, los botones de opción por lo general aparecen como un **grupo**, en el cual sólo *un* botón de opción puede estar seleccionado en un momento dado (vea la salida de la figura 12.20). Los botones de opción se utilizan para representar **opciones mutuamente excluyentes** (es decir, en un grupo *no* pueden seleccionarse varias opciones al mismo tiempo). La relación lógica entre los botones de opción se mantiene mediante un objeto **`ButtonGroup`** (paquete `javax.swing`), el cual en sí *no* es un componente de la GUI. Un objeto `ButtonGroup` organiza un grupo de botones y *no* se muestra a sí mismo en una interfaz de usuario. En vez de ello, se muestra en la GUI cada uno de los objetos `JRadioButton` del grupo.

La aplicación de las figuras 12.19 y 12.20 es similar a la de las figuras 12.17 y 12.18. El usuario puede alterar el estilo de letra de un objeto `JTextField`. La aplicación utiliza botones de opción que permiten que se seleccione solamente un estilo de letra en el grupo a la vez. La clase `PruebaBotonOpcion` (figura 12.20) contiene el método `main` que ejecuta esta aplicación.

```
1  // Fig. 12.19: MarcoBotonOpcion.java
2  // Creación de botones de opción, usando ButtonGroup y JRadioButton.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class MarcoBotonOpcion extends JFrame
13 {
14    private final JTextField campoTexto; // se utiliza para mostrar los cambios en
                                           el tipo de letra
15    private final Font tipoLetraSimple; // tipo de letra para texto simple
16    private final Font tipoLetraNegrita; // tipo de letra para texto en negrita
17    private final Font tipoLetraCursiva; // tipo de letra para texto en cursiva
18    private final Font tipoLetraNegritaCursiva; // tipo de letra para texto en
                                                  negrita y cursiva
19    private final JRadioButton simpleJRadioButton; // selecciona texto simple
20    private final JRadioButton negritaJRadioButton; // selecciona texto en negrita
21    private final JRadioButton cursivaJRadioButton; // selecciona texto en cursiva
22    private final JRadioButton negritaCursivaJRadioButton; // negrita y cursiva
23    private ButtonGroup grupoOpciones; // contiene los botones de opción
24
25    // El constructor de MarcoBotonOpcion agrega los objetos JRadioButton a JFrame
26    public MarcoBotonOpcion()
27    {
28       super("Prueba de RadioButton");
29       setLayout(new FlowLayout());
30
31       campoTexto = new JTextField("Observe el cambio en el estilo del tipo de
                                     letra", 25);
32       add(campoTexto); // agrega campoTexto a JFrame
33
34       // crea los botones de opción
35       simpleJRadioButton = new JRadioButton("Simple", true);
36       negritaJRadioButton = new JRadioButton("Negrita", false);
37       cursivaJRadioButton = new JRadioButton("Cursiva", false);
38       negritaCursivaJRadioButton = new JRadioButton("Negrita/Cursiva", false);
39       add(simpleJRadioButton); // agrega botón simple a JFrame
40       add(negritaJRadioButton); // agrega botón negrita a JFrame
41       add(cursivaJRadioButton); // agrega botón cursiva a JFrame
42       add(negritaCursivaJRadioButton); // agrega botón negrita y cursiva
43
44       // crea una relación lógica entre los objetos JRadioButton
45       grupoOpciones = new ButtonGroup(); // crea ButtonGroup
46       grupoOpciones.add(simpleJRadioButton); // agrega simple al grupo
47       grupoOpciones.add(negritaJRadioButton); // agrega negrita al grupo
48       grupoOpciones.add(cursivaJRadioButton); // agrega cursiva al grupo
49       grupoOpciones.add(negritaCursivaJRadioButton); // agrega negrita y cursiva
50
51       // crea objetos tipo de letra
52       tipoLetraSimple = new Font("Serif", Font.PLAIN, 14);
```

Fig. 12.19 | Creación de botones de opción, usando `ButtonGroup` y `JRadioButton` (parte 1 de 2).

```
53          tipoLetraNegrita = new Font("Serif", Font.BOLD, 14);
54          tipoLetraCursiva = new Font("Serif", Font.ITALIC, 14);
55          tipoLetraNegritaCursiva = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
56          campoTexto.setFont(tipoLetraSimple);
57
58          // registra eventos para los objetos JRadioButton
59          simpleJRadioButton.addItemListener(
60             new ManejadorBotonOpcion(tipoLetraSimple));
61          negritaJRadioButton.addItemListener(
62             new ManejadorBotonOpcion(tipoLetraNegrita));
63          cursivaJRadioButton.addItemListener(
64             new ManejadorBotonOpcion(tipoLetraCursiva));
65          negritaCursivaJRadioButton.addItemListener(
66             new ManejadorBotonOpcion(tipoLetraNegritaCursiva));
67       }
68
69       // clase interna privada para manejar eventos de botones de opción
70       private class ManejadorBotonOpcion implements ItemListener
71       {
72          private Font tipoLetra; // tipo de letra asociado con este componente
                                    de escucha
73
74          public ManejadorBotonOpcion(Font f)
75          {
76             tipoLetra = f;
77          }
78
79          // maneja los eventos de botones de opción
80          @Override
81          public void itemStateChanged(ItemEvent evento)
82          {
83             campoTexto.setFont(tipoLetra);
84          }
85       }
86    } // fin de la clase MarcoBotonOpcion
```

Fig. 12.19 | Creación de botones de opción, usando `ButtonGroup` y `JRadioButton` (parte 2 de 2).

```
1  // Fig. 12.20: PruebaBotonOpcion.java
2  // Prueba de MarcoBotonOpcion.
3  import javax.swing.JFrame;
4
5  public class PruebaBotonOpcion
6  {
7     public static void main(String[] args)
8     {
9        MarcoBotonOpcion marcoBotonOpcion = new MarcoBotonOpcion();
10       marcoBotonOpcion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoBotonOpcion.setSize(300, 100);
12       marcoBotonOpcion.setVisible(true);
13    }
14 } // fin de la clase PruebaBotonOpcion
```

Fig. 12.20 | Prueba de `MarcoBotonOpcion` (parte 1 de 2).

Fig. 12.20 | Prueba de `MarcoBotonOpcion` (parte 2 de 2).

En las líneas 35 a 42 del constructor (figura 12.19) se crean cuatro objetos `JRadioButton` y se agregan al objeto `JFrame`. Cada objeto `JRadioButton` se crea con una llamada al constructor como la de la línea 35. Este constructor especifica la etiqueta que aparece de manera predeterminada a la derecha del objeto `JRadioButton`, junto con su estado inicial. Un segundo argumento `true` indica que el objeto `JRadioButton` debe aparecer *seleccionado* al mostrarlo en pantalla.

En la línea 45 se instancia un objeto `ButtonGroup` llamado `grupoOpciones`. Este objeto es el "pegamento" que forma la relación lógica entre los cuatro objetos `JRadioButton` y permite que se seleccione solamente uno de los cuatro en un momento dado. Es posible que no se seleccione ningún `JRadioButton` en un `ButtonGroup`, pero esto *sólo* puede ocurrir si *no* se agregan objetos `JRadioButton` preseleccionados al objeto `ButtonGroup`, y si el usuario *no* ha seleccionado todavía un objeto `JRadioButton`. En las líneas 46 a 49 se utiliza el método **add** de `ButtonGroup` para asociar cada uno de los objetos `JRadioButton` con `grupoOpciones`. Si se agrega al grupo más de un objeto `JRadioButton` seleccionado, el *primer* objeto seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.

Los objetos `JRadioButton`, al igual que los objetos `JCheckbox`, generan eventos tipo `ItemEvent` cuando se *hace clic sobre ellos*. En las líneas 59 a 66 se crean cuatro instancias de la clase interna `ManejadorBotonOpcion` (declarada en las líneas 70 a 85). En este ejemplo, cada objeto componente de escucha de eventos se registra para manejar el evento `ItemEvent` que se genera cuando el usuario hace clic en cualquiera de los objetos `JRadioButton`. Observe que cada objeto `ManejadorBotonOpcion` se inicializa con un objeto `Font` específico (creado en las líneas 52 a 55).

La clase `ManejadorBotonOpcion` (línea 70 a 85) implementa la interfaz `ItemListener` para poder manejar los eventos `ItemEvent` generados por los objetos `JRadioButton`. El constructor almacena el objeto `Font` que recibe como un argumento en la variable de instancia `tipoLetra` (declarada en la línea 72). Cuando el usuario hace clic en un objeto `JRadioButton`, `grupoOpciones` desactiva el objeto `JRadioButton` previamente seleccionado y el método `itemStateChanged` (líneas 80 a 84) establece el tipo de letra en el objeto `JTextField` al tipo de letra almacenado en el objeto componente de escucha de eventos correspondiente al objeto `JRadioButton`. Observe que la línea 82 de la clase interna `ManejadorBotonOpcion` utiliza la variable de instancia `campoTexto` de la clase de nivel superior para establecer el tipo de letra.

12.11 JComboBox: uso de una clase interna anónima para el manejo de eventos

Un cuadro combinado (algunas veces conocido como **lista desplegable**) permite al usuario seleccionar *un* elemento de una lista (figura 12.22). Los cuadros combinados se implementan con la clase **JComboBox**, la

cual extiende a la clase `JComponent`. Ésta es una clase genérica, como la clase `ArrayList` (capítulo 7). Al crear un objeto `JComboBox` hay que especificar el tipo de los objetos que maneja; después el `JComboBox` muestra una representación `String` de cada objeto.

```
1  // Fig. 12.21: MarcoCuadroCombinado.java
2  // Objeto JComboBox que muestra una lista de nombres de imágenes.
3  import java.awt.FlowLayout;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JComboBox;
9  import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class MarcoCuadroCombinado extends JFrame
13 {
14    private final JComboBox<String> imagenesJComboBox; // contiene los nombres
                                                          de los iconos
15    private final JLabel etiqueta; // muestra el icono seleccionado
16
17    private static final String nombres[] =
18      {"insecto1.gif", "insecto2.gif",  "insectviaje.gif", "insectanim.gif"};
19    private final Icon[] iconos = {
20       new ImageIcon(getClass().getResource(nombres[0])),
21       new ImageIcon(getClass().getResource(nombres[1])),
22       new ImageIcon(getClass().getResource(nombres[2])),
23       new ImageIcon(getClass().getResource(nombres[3]))};
24
25    // El constructor de MarcoCuadroCombinado agrega un objeto JComboBox a JFrame
26    public MarcoCuadroCombinado()
27    {
28       super("Prueba de JComboBox");
29       setLayout(new FlowLayout()); // establece el esquema del marco
30
31       imagenesJComboBox = new JComboBox<String>(nombres); // establece JComboBox
32       imagenesJComboBox.setMaximumRowCount(3); // muestra tres filas
33
34       imagenesJComboBox.addItemListener(
35          new ItemListener() // clase interna anónima
36          {
37             // maneja evento de JComboBox
38             @Override
39             public void itemStateChanged(ItemEvent evento)
40             {
41                // determina si está seleccionado el elemento
42                if (evento.getStateChange() == ItemEvent.SELECTED)
43                   etiqueta.setIcon(iconos[
44                      imagenesJComboBox.getSelectedIndex()]);
45             }
46          } // fin de la clase interna anónima
47       ); // fin de la llamada a addItemListener
48
```

Fig. 12.21 | Objeto `JComboBox` que muestra una lista de nombres de imágenes (parte 1 de 2).

```
49          add(imagenesJComboBox); // agrega cuadro combinado a JFrame
50          etiqueta = new JLabel(iconos[0]); // muestra el primer icono
51          add(etiqueta); // agrega etiqueta a JFrame
52       }
53    } // fin de la clase MarcoCuadroCombinado
```

Fig. 12.21 | Objeto `JComboBox` que muestra una lista de nombres de imágenes (parte 2 de 2).

```
1  // Fig. 12.22: PruebaCuadroCombinado.java
2  // Prueba de MarcoCuadroCombinado.
3  import javax.swing.JFrame;
4
5  public class PruebaCuadroCombinado
6  {
7     public static void main(String[] args)
8     {
9        MarcoCuadroCombinado marcoCuadroCombinado = new MarcoCuadroCombinado();
10       marcoCuadroCombinado.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoCuadroCombinado.setSize(350, 150);
12       marcoCuadroCombinado.setVisible(true);
13    }
14 } // fin de la clase PruebaCuadroCombinado
```

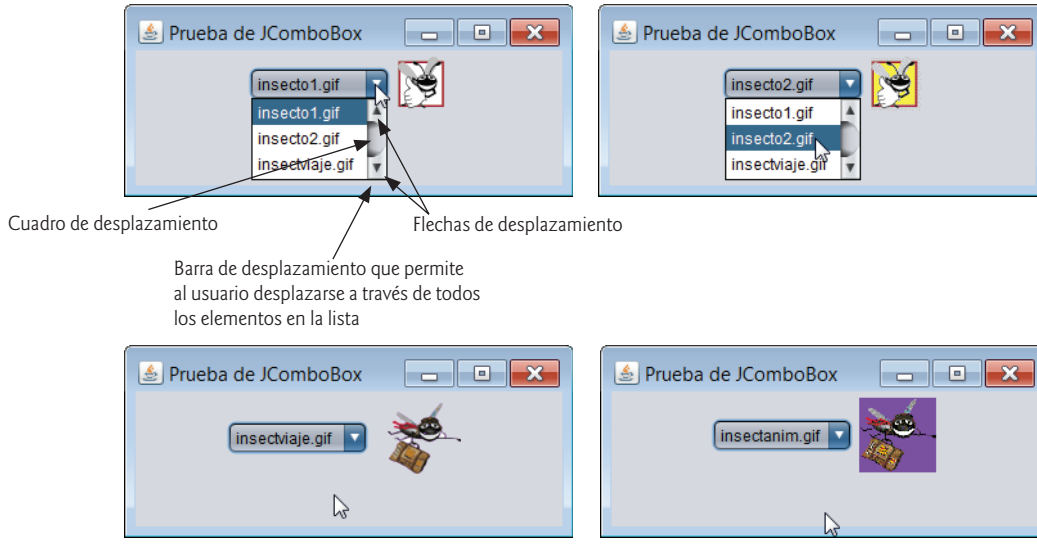


Fig. 12.22 | Prueba de `MarcoCuadroCombinado`.

Los objetos `JComboBox` generan eventos `ItemEvent`, al igual que los objetos `JCheckBox` y `JRadioButton`. Este ejemplo también demuestra una forma especial de clase interna, que se utiliza con frecuencia en el manejo de eventos. La aplicación (figuras 12.21 y 12.22) utiliza un objeto `JComboBox` para proporcionar una lista de cuatro nombres de archivos de imágenes, de los cuales el usuario puede seleccionar una imagen para mostrarla en pantalla. Cuando el usuario selecciona un nombre, la aplicación muestra la imagen correspondiente como un objeto `Icon` en un objeto `JLabel`. La clase `PruebaCuadroCombinado` (figura 12.22) contiene el método `main` que ejecuta esta aplicación. Las capturas de pantalla para esta aplicación muestran la lista `JComboBox` después de hacer una selección, para ilustrar cuál nombre de archivo de imagen fue seleccionado.

En las líneas 19 a 23 (figura 12.21) se declara e inicializa el arreglo `iconos` con cuatro nuevos objetos `ImageIcon`. El arreglo `String` llamado `nombres` (líneas 17 y 18) contiene los nombres de los cuatro archivos de imagen que se guardan en el mismo directorio que la aplicación.

En la línea 31, el constructor inicializa un objeto `JComboBox` utilizando los objetos `String` en el arreglo `nombres` como los elementos en la lista. Cada elemento de la lista tiene un **índice**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado al mostrar el objeto `JComboBox`. Los otros elementos se seleccionan haciendo clic en el objeto `JComboBox`, y después seleccionando un elemento de la lista que aparece.

En la línea 32 se utiliza el método **`setMaximumRowCount`** de `JComboBox` para establecer el máximo número de elementos a mostrar cuando el usuario haga clic en el objeto `JComboBox`. Si hay elementos adicionales, el objeto `JComboBox` proporciona una **barra de desplazamiento** (vea la primera captura de pantalla) que permite al usuario desplazarse por todos los elementos en la lista. El usuario puede hacer clic en las **flechas de desplazamiento** que están en las partes superior e inferior de la barra de desplazamiento para avanzar hacia arriba y hacia abajo de la lista, un elemento a la vez, o puede arrastrar hacia arriba y hacia abajo el **cuadro de desplazamiento** que está en medio de la barra de desplazamiento. Para arrastrar el cuadro de desplazamiento, posicione el ratón sobre éste, mantenga presionado el botón izquierdo y mueva el ratón. En este ejemplo, la lista desplegable es demasiado corta como para poder arrastrar el cuadro de desplazamiento, por lo que puede hacer clic en las flechas arriba y abajo o usar la rueda de su ratón para desplazarse por los cuatro elementos en la lista. La línea 49 adjunta el objeto `JComboBox` al esquema `FlowLayout` de `MarcoCuadroCombinado` (que se establece en la línea 29). La línea 50 crea el objeto `JLabel` que muestra objetos `ImageIcon` y lo inicializa con el primer objeto `ImageIcon` en el arreglo `iconos`. La línea 51 adjunta el objeto `JLabel` al esquema `FlowLayout` de `MarcoCuadroCombinado`.

Observación de apariencia visual 12.12

Establezca el número máximo de filas en un objeto `JComboBox` a un valor que evite que la lista se expanda fuera de los límites de la ventana en la que se utilice.

Uso de una clase interna anónima para el manejo de eventos

Las líneas 34 a 46 representan una instrucción que declara la clase del componente de escucha de eventos, crea un objeto de esa clase y registra el objeto como el componente de escucha para los eventos `ItemEvent` de `imagenesJComboBox`. Este objeto componente de escucha de eventos es una instancia de una **clase interna anónima**; una clase interna que se declara sin un nombre y por lo general aparece dentro de la declaración de un método. *Al igual que las demás clases internas, una clase interna anónima puede acceder a los miembros de su clase de nivel superior*. Sin embargo, una clase interna anónima tiene acceso limitado a las variables locales del método en el que está declarada. Como una clase interna anónima no tiene nombre, debe crearse un objeto de la misma en el punto en el que se declara la clase (empezando en la línea 35).

Observación de ingeniería de software 12.3

Una clase interna anónima declarada en un método puede acceder a las variables de instancia y los métodos del objeto de la clase de nivel superior que la declaró, así como a las variables locales `final` del método, pero no puede acceder a las variables locales no `final` del método. A partir de Java SE 8, las clases internas anónimas también pueden acceder a las variables locales "efectivamente `final`" del método; vea el capítulo 17 para más información.

Las líneas 34 a 47 son una llamada al método `addItemListener` de `imagenesJComboBox`. El argumento para este método debe ser un objeto que *sea un* `ItemListener` (es decir, cualquier objeto de una clase que implemente a `ItemListener`). Las líneas 35 a 46 son una expresión de creación de instancias de clase que declara una clase interna anónima y crea un objeto de esa clase. Después se pasa una referencia a ese

objeto como argumento para `addItemListener`. La sintaxis `ItemListener()` después de `new` empieza la declaración de una clase interna anónima que implementa a la interfaz `ItemListener`. Esto es similar a empezar una declaración con

```
public class MiManejador implements ItemListener
```

La llave izquierda de apertura en la línea 36 y la llave derecha de cierre en la línea 46 delimitan el cuerpo de la clase interna anónima. Las líneas 38 a 45 declaran el método `itemStateChanged` de `ItemListener`. Cuando el usuario hace una selección de `imagenesJComboBox`, este método establece el objeto `Icon` de `etiqueta`. El objeto `Icon` se selecciona del arreglo `iconos`, determinando el índice del elemento seleccionado en el objeto `JComboBox` con el método **`getSelectedIndex`** en la línea 44. Para cada elemento seleccionado de un `JComboBox`, primero se deselecciona otro elemento; por lo tanto, ocurren dos eventos tipo `ItemEvent` cuando se selecciona un elemento. Deseamos mostrar sólo el icono para el elemento que el usuario acaba de seleccionar. Por esta razón, la línea 42 determina si el método **`getStateChange`** de `ItemEvent` devuelve `ItemEvent.SELECTED`. De ser así, las líneas 43 y 44 establecen el icono de `etiqueta`.

Observación de ingeniería de software 12.4

Al igual que cualquier otra clase, cuando una clase interna anónima implementa a una interfaz, la clase debe implementar todos los métodos en la interfaz.

La sintaxis que se muestra en las líneas 35 a 46 para crear un manejador de eventos con una clase interna anónima es similar al código que genera un entorno de desarrollo integrado (IDE) de Java. Por lo general, un IDE permite al programador diseñar una GUI en forma visual, y después el IDE genera código que implementa a la GUI. El programador sólo inserta instrucciones en los métodos manejadores de eventos que declaran cómo manejar cada evento.

Java SE 8: Cómo implementar clases internas anónimas con lambdas

En la sección 17.9 le mostraremos cómo usar lambdas de Java SE 8 para crear manejadores de eventos. Como aprenderá, el compilador traduce una lambda en un objeto de una clase interna anónima.

12.12 JList

Una lista muestra una serie de elementos, de la cual el usuario puede *seleccionar uno o más elementos* (vea la salida de la figura 12.24). Las listas se crean con la clase `JList`, que extiende directamente a la clase `JComponent`. La clase `JList` (que, al igual que `JComboBox`, es una clase genérica) soporta **listas de selección simple** (que permiten seleccionar solamente un elemento a la vez) y **listas de selección múltiple** (que permiten seleccionar cualquier número de elementos a la vez). En esta sección hablaremos sobre las listas de selección simple.

La aplicación de las figuras 12.23 y 12.24 crea un objeto `JList` que contiene los nombres de 13 colores. Al hacer clic en el nombre de un color en el objeto `JList`, ocurre un evento `ListSelectionEvent` y la aplicación cambia el color de fondo de la ventana de aplicación al color seleccionado. La clase `PruebaLista` (figura 12.24) contiene el método `main` que ejecuta esta aplicación.

```
1 // Fig. 12.23: MarcoLista.java
2 // Objeto JList que muestra una lista de colores.
3 import java.awt.FlowLayout;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6 import javax.swing.JList;
```

Fig. 12.23 | Objeto `JList` que muestra una lista de colores (parte 1 de 2).

```
7  import javax.swing.JScrollPane;
8  import javax.swing.event.ListSelectionListener;
9  import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class MarcoLista extends JFrame
13 {
14    private final JList<String> listaJListColores; // lista para mostrar colores
15    private static final String[] nombresColores = {"Negro", "Azul", "Cyan",
16       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta",
17       "Naranja", "Rosa", "Rojo", "Blanco", "Amarillo"};
18    private static final Color[] colores = {Color.BLACK, Color.BLUE,
19       Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20       Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21       Color.RED, Color.WHITE, Color.YELLOW};
22
23    // El constructor de MarcoLista agrega a JFrame el JScrollPane que contiene a
         JList
24    public MarcoLista()
25    {
26       super("Prueba de JList");
27       setLayout(new FlowLayout());
28
29       listaJListColores = new JList<String>(nombresColores); // lista de
                                                                   nombresColores
30       listaJListColores.setVisibleRowCount(5); // muestra cinco filas a la vez
31
32       // no permite selecciones múltiples
33       listaJListColores.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
34
35       // agrega al marco un objeto JScrollPane que contiene a JList
36       add(new JScrollPane(listaJListColores));
37
38       listaJListColores.addListSelectionListener(
39          new ListSelectionListener() // clase interna anónima
40          {
41             // maneja los eventos de selección de la lista
42             @Override
43             public void valueChanged(ListSelectionEvent evento)
44             {
45                getContentPane().setBackground(
46                   colores[listaJListColores.getSelectedIndex()]);
47             }
48          }
49       );
50    }
51 } // fin de la clase MarcoLista
```

Fig. 12.23 | Objeto `JList` que muestra una lista de colores (parte 2 de 2).

```
1  // Fig. 12.24: PruebaLista.java
2  // Selección de colores de un objeto JList.
3  import javax.swing.JFrame;
4
```

Fig. 12.24 | Seleccionar colores de un objeto `JList` (parte 1 de 2).

```
5  public class PruebaLista
6  {
7     public static void main(String[] args)
8     {
9        MarcoLista marcoLista = new MarcoLista(); // crea objeto MarcoLista
10       marcoLista.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoLista.setSize(350, 150);
12       marcoLista.setVisible(true);
13    }
14 } // fin de la clase PruebaLista
```

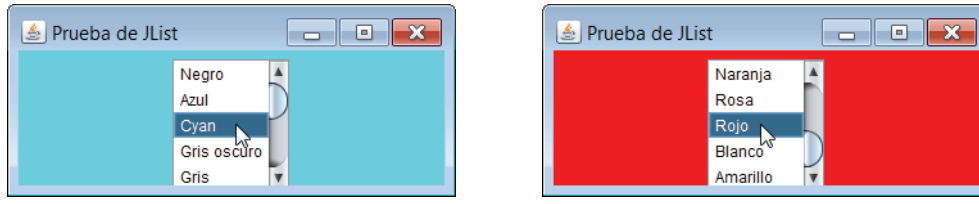


Fig. 12.24 | Seleccionar colores de un objeto `JList` (parte 2 de 2).

La línea 29 (figura 12.23) crea el objeto `listaJListColores` llamado `JList`. El argumento para el constructor de `JList` es el arreglo de objetos `Object` (en este caso, objetos `String`) a mostrar en la lista. La línea 30 utiliza el método **`setVisibleRowCount`** de `JList` para determinar el número de elementos *visibles* en la lista.

La línea 33 utiliza el método **`setSelectionMode`** de `JList` para especificar el **modo de selección** de la lista. La clase **`ListSelectionModel`** (del paquete `javax.swing`) declara tres constantes que especifican el modo de selección de un objeto `JList`: **`SINGLE_SELECTION`** (que sólo permite seleccionar un elemento a la vez), **`SINGLE_INTERVAL_SELECTION`** (para una lista de selección múltiple que permite seleccionar varios elementos contiguos) y **`MULTIPLE_INTERVAL_SELECTION`** (para una lista de selección múltiple que no restringe los elementos que se pueden seleccionar).

A diferencia de un objeto `JComboBox`, un objeto `JList` *no proporciona una barra de desplazamiento* si hay más elementos en la lista que el número de filas visibles. En este caso se utiliza un objeto **`JScrollPane`** para proporcionar la capacidad de desplazamiento. En la línea 36 se agrega una nueva instancia de la clase `JScrollPane` al objeto `JFrame`. El constructor de `JScrollPane` recibe como argumento el objeto `JComponent` que necesita funcionalidad de desplazamiento (en este caso, `listaJListColores`). Observe en las capturas de pantalla que aparece una barra de desplazamiento creada por el objeto `JScrollPane` en el lado derecho del objeto `JList`. De manera predeterminada, la barra de desplazamiento sólo aparece cuando el número de elementos en el objeto `JList` excede al número de elementos visibles.

Las líneas 38 a 49 usan el método **`addListSelectionListener`** de `JList` para registrar un objeto que implementa a **`ListSelectionListener`** (paquete `javax.swing.event`) como el componente de escucha para los eventos de selección de `JList`. Una vez más, utilizamos una instancia de una clase interna anónima (líneas 39 a 48) como el componente de escucha. En este ejemplo, cuando el usuario realiza una selección de `listaJListColores`, el método **`valueChanged`** (línea 42 a 47) debería cambiar el color de fondo del objeto `MarcoLista` al color seleccionado. Esto se logra en las líneas 45 y 46. Observe el uso del método **`getContentPane`** de `JFrame` en la línea 45. Cada objeto `JFrame` en realidad consiste de *tres niveles*: el *fondo*, el *panel de contenido* y el *panel de vidrio*. El panel de contenido aparece enfrente del fondo, y es en donde se muestran los componentes de la GUI en el objeto `JFrame`. El panel de vidrio se utiliza para mostrar cuadros de información sobre herramientas y otros elementos que deben aparecer enfrente de los componentes de la GUI en la pantalla. El panel de contenido oculta por completo el fondo del objeto `JFrame`; por ende, para cambiar el color de fondo detrás de los componentes de la GUI, debe cambiar el color de fondo del panel de contenido. El método `getContentPane` devuelve una

referencia al panel de contenido del objeto `JFrame` (un objeto de la clase `Container`). En la línea 45, usamos esa referencia para llamar al método **`setBackground`**, el cual establece el color de fondo del panel de contenido a un elemento en el arreglo `colores`. El color se selecciona del arreglo mediante el uso del índice del elemento seleccionado. El método **`getSelectedItem`** de `JList` devuelve el índice del elemento seleccionado. Al igual que con los arreglos y los objetos `JComboBox`, los índices en los objetos `JList` están basados en cero.

12.13 Listas de selección múltiple

Una **lista de selección múltiple** permite al usuario seleccionar varios elementos de un objeto `JList` (vea la salida de la figura 12.26). Una lista `SINGLE_INTERVAL_SELECTION` permite la selección de un rango contiguo de elementos. Para ello, haga clic en el primer elemento y después oprima (y mantenga oprimida) la tecla *Mayús* mientras hace clic en el último elemento a seleccionar en el rango. Una lista `MULTIPLE_INTERVAL_SELECTION` (la opción predeterminada) permite una selección de rango continuo, como se describe para una lista `SINGLE_INTERVAL_SELECTION`. Dicha lista también permite que se seleccionen diversos elementos, oprimiendo y manteniendo oprimida la tecla *Ctrl* mientras hace clic en cada elemento a seleccionar. Para deseleccionar un elemento, oprima y mantenga oprimida la tecla *Ctrl* mientras hace clic en el elemento por segunda vez.

La aplicación de las figuras 12.25 y 12.26 utiliza listas de selección múltiple para copiar elementos de un objeto `JList` a otro. Una lista es de tipo `MULTIPLE_INTERVAL_SELECTION` y la otra es de tipo `SINGLE_INTERVAL_SELECTION`. Cuando ejecute la aplicación, trate de usar las técnicas de selección descritas anteriormente para seleccionar elementos en ambas listas.

```
1  // Fig. 12.25: MarcoSeleccionMultiple.java
2  // Objeto JList que permite selecciones múltiples.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JList;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MarcoSeleccionMultiple extends JFrame
13 {
14    private final JList<String> listaJListColores; // lista para guardar los nombres
                                                      de los colores
15    private final JList<String> listaJListCopia; // lista en la que se van a copiar
                                                    los nombres de los colores
16    private JButton botonJButtonCopiar; // botón para copiar los nombres seleccionados
17    private static final String[] nombresColores = {"Negro", "Azul", "Cyan",
18       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta", "Naranja",
19       "Rosa", "Rojo", "Blanco", "Amarillo"};
20
21    // Constructor de MarcoSeleccionMultiple
22    public MarcoSeleccionMultiple()
23    {
24       super("Listas de seleccion multiple");
25       setLayout(new FlowLayout());
26
27       listaJListColores = new JList<String>(nombresColores); // lista de nombres
                                                                  de colores
```

Fig. 12.25 | Objeto `JList` que permite selecciones múltiples (parte 1 de 2).

```
28          listaJListColores.setVisibleRowCount(5); // muestra cinco filas
29          listaJListColores.setSelectionMode(
30             ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
31          add(new JScrollPane(listaJListColores)); // agrega lista con panel de
                                                     desplazamiento
32
33          botonJButtonCopiar = new JButton("Copiar >>>");
34          botonJButtonCopiar.addActionListener(
35             new ActionListener() // clase interna anónima
36             {
37                // maneja evento de botón
38                @Override
39                public void actionPerformed(ActionEvent evento)
40                {
41                   // coloca los valores seleccionados en listaJListCopia
42                   listaJListCopia.setListData(
43                      listaJListColores.getSelectedValuesList().toArray(
44                         new String[0]));
45                }
46             }
47          );
48
49          add(botonJButtonCopiar); // agrega el botón copiar a JFrame
50
51          listaJListCopia = new JList<String>(); // lista para guardar nombres de
                                                   colores copiados
52          listaJListCopia.setVisibleRowCount(5); // muestra 5 filas
53          listaJListCopia.setFixedCellWidth(100); // establece la anchura
54          listaJListCopia.setFixedCellHeight(15); // establece la altura
55          listaJListCopia.setSelectionMode(
56             ListSelectionModel.SINGLE_INTERVAL_SELECTION);
57          add(new JScrollPane(listaJListCopia)); // agrega lista con panel de
                                                   desplazamiento
58       }
59    } // fin de la clase MarcoSeleccionMultiple
```

Fig. 12.25 | Objeto `JList` que permite selecciones múltiples (parte 2 de 2).

```
1  // Fig. 12.26: PruebaSeleccionMultiple.java
2  // Prueba de MarcoSeleccionMultiple.
3  import javax.swing.JFrame;
4
5  public class PruebaSeleccionMultiple
6  {
7     public static void main(String[] args)
8     {
9        MarcoSeleccionMultiple marcoSeleccionMultiple =
10          new MarcoSeleccionMultiple();
11       marcoSeleccionMultiple.setDefaultCloseOperation(
12          JFrame.EXIT_ON_CLOSE);
13       marcoSeleccionMultiple.setSize(350, 140);
14       marcoSeleccionMultiple.setVisible(true);
15    }
16 } // fin de la clase PruebaSeleccionMultiple
```

Fig. 12.26 | Prueba de `MarcoSeleccionMultiple` (parte 1 de 2).

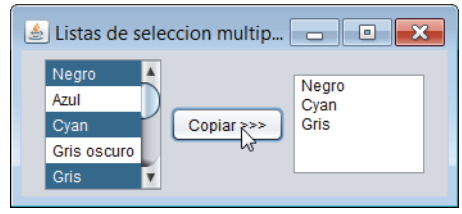


Fig. 12.26 | Prueba de `MarcoSeleccionMultiple` (parte 2 de 2).

En la línea 27 de la figura 12.25 se crea el objeto `JList` llamado `listaJListColores` y se inicializa con las cadenas en el arreglo `nombresColores`. En la línea 28 se establece el número de filas visibles en `listaJListColores` a 5. En las líneas 29 y 30 se especifica que `listaJListColores` es una lista de tipo `MULTIPLE_INTERVAL_SELECTION`. En la línea 31 se agrega un nuevo objeto `JScrollPane`, que contiene `listaJListColores`, al panel `JFrame`. En las líneas 51 a 57 se realizan tareas similares para `listaJListCopia`, la cual se declara como una lista tipo `SINGLE_INTERVAL_SELECTION`. Si un objeto `JList` no contiene elementos, no se mostrará en un esquema `FlowLayout`. Por esta razón, en las líneas 53 y 54 se utilizan los métodos **`setFixedCellWidth`** y **`setFixedCellHeigth`** de `JList` para establecer la anchura de `listaJListCopia` en 100 píxeles y la altura de cada elemento en el objeto `JList` a 15 píxeles, respectivamente.

Por lo general, un evento generado por otro componente de la GUI (lo que se conoce como un **evento externo**) especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`. En este ejemplo, el usuario hace clic en el objeto `JButton` llamado `botonJButtonCopiar` para desencadenar el evento que copia los elementos seleccionados en `listaJListColores` a `listaJListCopia`.

Las líneas 34 a 47 declaran, crean y registran un objeto `ActionListener` para el objeto `botonJButtonCopiar`. Cuando el usuario hace clic en `botonJButtonCopiar`, el método `actionPerformed` (líneas 38 a 45) utiliza el método **`setListData`** de `JList` para establecer los elementos mostrados en `listaJListCopia`. En las líneas 43 y 44 se hace una llamada al método **`getSelectedValues`** de `listaJListColores`, el cual devuelve un objeto `List<String>` (debido a que el objeto `JList` se creó como `JList<String>`) que representa a los elementos seleccionados en `listaJListColores`. Llamamos al método `toArray` de `List<String>` para convertir esto en un arreglo de objetos `String` que puedan pasarse como el argumento para el método `setListData` de `listaJListCopia`. El método `toArray` de `List` recibe como su argumento un arreglo que representa el tipo de arreglo que devolverá el método. Aprenderá más sobre `List` y `toArray` en el capítulo 16.

Tal vez se pregunte por qué puede usarse `listaJListCopia` en la línea 42, aun cuando la aplicación no crea el objeto al cual hace referencia sino hasta la línea 49. Recuerde que el método `actionPerformed` (líneas 38 a 45) no se ejecuta sino hasta que el usuario oprime el botón `botonJButtonCopiar`, lo cual no puede ocurrir sino hasta que el constructor termine su ejecución y la aplicación muestre la GUI. En ese punto en la ejecución de la aplicación, `listaJListCopia` ya se ha inicializado con un nuevo objeto `JList`.

12.14 Manejo de eventos de ratón

En esta sección presentaremos las interfaces de escucha de eventos **`MouseListener`** y **`MouseMotionListener`** para manejar **eventos de ratón**. Estos eventos pueden procesarse para cualquier componente de la GUI que se derive de `java.awt.Component`. Los métodos de las interfaces `MouseListener` y `MouseMotionListener` se sintetizan en la figura 12.27. El paquete `javax.swing.event` contiene la interfaz **`MouseInputListener`**, la cual extiende a las interfaces `MouseListener` y `MouseMotionListener` para crear una sola interfaz que contiene todos los métodos de `MouseListener` y `MouseMotionListener`. Estos métodos se llaman cuando el ratón interactúa con un objeto `Component`, si se registran objetos componentes de escucha de eventos apropiados para ese objeto `Component`.

Cada uno de los métodos manejadores de eventos de ratón recibe un objeto **MouseEvent** como su argumento, el cual contiene información acerca del evento de ratón que ocurrió, incluyendo las coordenadas *x* y *y* de su ubicación. Estas coordenadas se miden desde la *esquina superior izquierda* del componente de la GUI en el que ocurrió el evento. Las coordenadas *x* empiezan en 0 y se *incrementan de izquierda a derecha*. Las coordenadas *y* empiezan en 0 y se *incrementan de arriba hacia abajo*. Los métodos y constantes de la clase **InputEvent** (superclase de `MouseEvent`) permiten a una aplicación determinar cuál fue el botón del ratón que oprimió el usuario.

Métodos de las interfaces `MouseListener` y `MouseMotionListener`

Métodos de la interfaz `MouseListener`

`public void mousePressed(MouseEvent evento)`

Es llamado cuando se *oprime* un botón del ratón, mientras el cursor del ratón está sobre un componente.

`public void mouseClicked(MouseEvent evento)`

Es llamado cuando se *oprime* y *suelta* un botón del ratón, mientras el cursor del ratón permanece estacionario sobre un componente. Este evento siempre va precedido por una llamada a `mousePressed` y `mouseReleased`.

`public void mouseReleased(MouseEvent evento)`

Es llamado cuando se *suelta un botón de ratón después de ser oprimido*. Este evento siempre va precedido por una llamada a `mousePressed` y por una o más llamadas a `mouseDragged`.

`public void mouseEntered(MouseEvent evento)`

Es llamado cuando el cursor del ratón *entra* a los límites de un componente.

`public void mouseExited(MouseEvent evento)`

Es llama cuando el cursor del ratón *sale* de los límites de un componente.

Métodos de la interfaz `MouseMotionListener`

`public void mouseDragged(MouseEvent evento)`

Es llamado cuando el botón del ratón se *oprime* mientras el cursor del ratón se encuentra sobre un componente y el ratón se *mueve* mientras el botón *sigue oprimido*. Este evento siempre va precedido por una llamada a `mousePressed`. Todos los eventos de arrastre del ratón se envían al componente en el cual empezó la acción de arrastre.

`public void mouseMoved(MouseEvent evento)`

Es llamado al *moverse* el ratón (sin oprimir los botones del ratón) cuando su cursor se encuentra sobre un componente. Todos los eventos de movimiento se envían al componente sobre el cual se encuentra el ratón posicionado en ese momento.

Fig. 12.27 | Métodos de las interfaces `MouseListener` y `MouseMotionListener`.

Observación de ingeniería de software 12.5

Las llamadas al método `mouseDragged` se envían al objeto `MouseMotionListener` para el objeto `Component` en el que empezó la operación de arrastre. De manera similar, la llamada al método `mouseReleased` al final de una operación de arrastre se envía al objeto `MouseListener` para el objeto `Component` en el que empezó la operación de arrastre.

Java también cuenta con la interfaz **MouseWheelListener** para permitir a las aplicaciones responder a la *rotación de la rueda de un ratón*. Esta interfaz declara el método **mouseWheelMoved**, el cual recibe un evento **MouseWheelEvent** como argumento. La clase MouseWheelEvent (una subclase de MouseEvent) contiene métodos que permiten al manejador de eventos obtener información acerca de la cantidad de rotación de la rueda.

Cómo rastrear eventos de ratón en un objeto JPanel

La aplicación RastreadorRaton (figuras 12.28 y 12.29) demuestra el uso de los métodos de las interfaces MouseListener y MouseMotionListener. La clase de manejador de evento (líneas 36 a 97 de la figura 12.28) implementa ambas interfaces, en cuyo caso usted *debe* declarar los siete métodos de estas dos interfaces. Cada evento de ratón en este ejemplo muestra un objeto String en el objeto JLabel llamado barraEstado, que está unido a la parte inferior de la ventana.

```
1  // Fig. 12.28: MarcoRastreadorRaton.java
2  // Manejo de eventos de ratón.
3  import java.awt.Color;
4  import java.awt.BorderLayout;
5  import java.awt.event.MouseListener;
6  import java.awt.event.MouseMotionListener;
7  import java.awt.event.MouseEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MarcoRastreadorRaton extends JFrame
13 {
14    private final JPanel panelRaton; // panel en el que ocurrirán los eventos
                                       de ratón
15    private final JLabel barraEstado; // muestra información de los eventos
16
17    // El constructor de MarcoRastreadorRaton establece la GUI y
18    // registra los manejadores de eventos de ratón
19    public MarcoRastreadorRaton()
20    {
21       super("Demostracion de los eventos de raton");
22
23       panelRaton = new JPanel();
24       panelRaton.setBackground(Color.WHITE);
25       add(panelRaton, BorderLayout.CENTER); // agrega el panel a JFrame
26
27       barraEstado = new JLabel("Raton fuera de JPanel");
28       add(barraEstado, BorderLayout.SOUTH); // agrega etiqueta a JFrame
29
30       // crea y registra un componente de escucha para los eventos de ratón
            y de su movimiento
31       ManejadorRaton manejador = new ManejadorRaton();
32       panelRaton.addMouseListener(manejador);
33       panelRaton.addMouseMotionListener(manejador);
34    }
35
```

Fig. 12.28 | Manejo de eventos de ratón (parte 1 de 3).

```
36    private class ManejadorRaton implements MouseListener,
37       MouseMotionListener
38    {
39       // Los manejadores de eventos de MouseListener
40       // manejan el evento cuando se suelta el ratón justo después de oprimir
            el botón
41       @Override
42       public void mouseClicked(MouseEvent evento)
43       {
44          barraEstado.setText(String.format("Se hizo clic en [%d, %d]",
45             evento.getX(), evento.getY()));
46       }
47
48       // maneja evento cuando se oprime el ratón
49       @Override
50       public void mousePressed(MouseEvent evento)
51       {
52          barraEstado.setText(String.format("Se oprimio en [%d, %d]",
53             evento.getX(), evento.getY()));
54       }
55
56       // maneja evento cuando se suelta el botón del ratón
57       @Override
58       public void mouseReleased(MouseEvent evento)
59       {
60          barraEstado.setText(String.format("Se solto en [%d, %d]",
61             evento.getX(), evento.getY()));
62       }
63
64       // maneja evento cuando el ratón entra al área
65       @Override
66       public void mouseEntered(MouseEvent evento)
67       {
68          barraEstado.setText(String.format("Raton entro en [%d, %d]",
69             evento.getX(), evento.getY()));
70          panelRaton.setBackground(Color.GREEN);
71       }
72
73       // maneja evento cuando el ratón sale del área
74       @Override
75       public void mouseExited(MouseEvent evento)
76       {
77          barraEstado.setText("Raton fuera de JPanel");
78          panelRaton.setBackground(Color.WHITE);
79       }
80
81       // Los manejadores de eventos de MouseMotionListener manejan
82       // el evento cuando el usuario arrastra el ratón con el botón oprimido
83       @Override
84       public void mouseDragged(MouseEvent evento)
85       {
86          barraEstado.setText(String.format("Se arrastro en [%d, %d]",
87             evento.getX(), evento.getY()));
88       }
```

Fig. 12.28 | Manejo de eventos de ratón (parte 2 de 3).

```
89
90          // maneja evento cuando el usuario mueve el ratón
91          @Override
92          public void mouseMoved(MouseEvent evento)
93          {
94             barraEstado.setText(String.format("Se movio en [%d, %d]",
95                evento.getX(), evento.getY()));
96          }
97       } // fin de la clase interna ManejadorRaton
98    } // fin de la clase MarcoRastreadorRaton
```

Fig. 12.28 | Manejo de eventos de ratón (parte 3 de 3).

```
1  // Fig. 12.29: MarcoRastreadorRaton.java
2  // Prueba de MarcoRastreadorRaton.
3  import javax.swing.JFrame;
4
5  public class RastreadorRaton
6  {
7     public static void main(String[] args)
8     {
9        MarcoRastreadorRaton marcoRastreadorRaton = new MarcoRastreadorRaton();
10       marcoRastreadorRaton.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoRastreadorRaton.setSize(300, 100);
12       marcoRastreadorRaton.setVisible(true);
13    }
14 } // fin de la clase RastreadorRaton
```

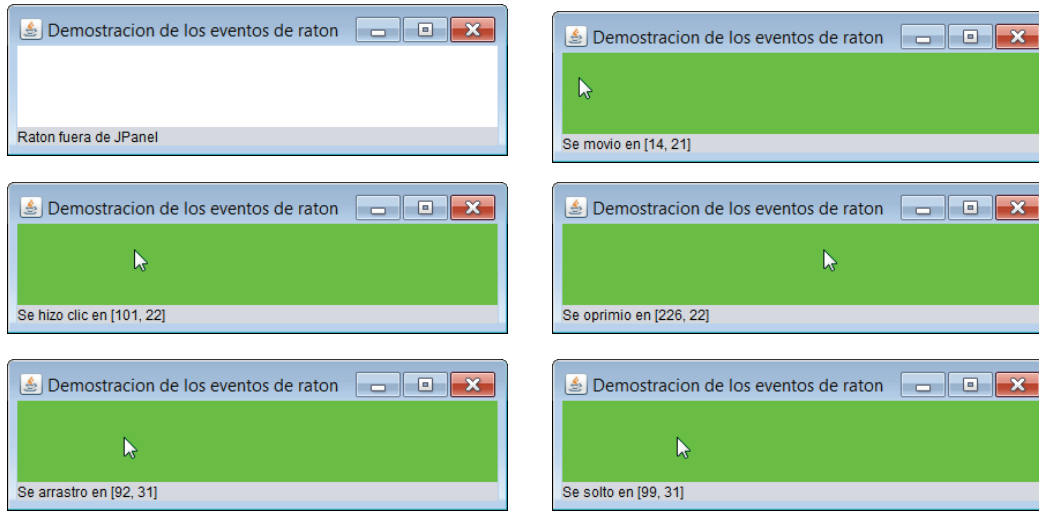


Fig. 12.29 | Prueba de `MarcoRastreadorRaton`.

La línea 23 crea el objeto `JPanel` llamado `panelRaton`. Los eventos de ratón de este objeto `JPanel` serán rastreados por la aplicación. En la línea 24 se establece el color de fondo de `panelRaton` en blanco. Cuando el usuario mueva el ratón hacia el `panelRaton`, la aplicación cambiará el color de fondo de `panelRaton` a verde. Cuando el usuario mueva el ratón hacia fuera del `panelRaton`, la aplicación cambiará el color de fondo de nuevo a blanco. En la línea 25 se adjunta el objeto `panelRaton` al objeto `JFrame`.

Como vimos, por lo general debemos especificar el esquema de los componentes de GUI en un objeto `JFrame`. En esa sección presentamos el administrador de esquemas `FlowLayout`. Aquí utilizamos el esquema predeterminado del panel de contenido de un objeto `JFrame`, **`BorderLayout`**, el cual ordena los componentes en cinco regiones: **`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`** y **`CENTER`**. `NORTH` corresponde a la parte superior del contenedor. Este ejemplo utiliza las regiones `CENTER` y `SOUTH`. En la línea 25 se utiliza una versión con dos argumentos del método `add` para colocar a `panelRaton` en la región `CENTER`. El esquema `BorderLayout` ajusta automáticamente el tamaño del componente en la región `CENTER` para utilizar todo el espacio en el objeto `JFrame` que no esté ocupado por los componentes de otras regiones. En la sección 12.18.2 hablaremos con más detalle sobre `BorderLayout`.

En las líneas 27 y 28 del constructor se declara el objeto `JLabel` llamado `barraEstado` y se adjunta a la región `SOUTH` del objeto `JFrame`. Este objeto `JLabel` ocupa la anchura del objeto `JFrame`. La altura de la región se determina con base en el objeto `Jlabel`.

En la línea 31 se crea una instancia de la clase interna `ManejadorRaton` (líneas 36 a 97) llamada `manejador`, la cual responde a los eventos de ratón. En las líneas 32 y 33 se registra `manejador` como el componente de escucha para los eventos de ratón de `panelRaton`. Los métodos **`addMouseListener`** y **`addMouseMotionListener`** se heredan indirectamente de la clase `Component`, y pueden utilizarse para registrar objetos `MouseListener` y `MouseMotionListener`, respectivamente. Un objeto `ManejadorRaton` *es un* `MouseListener` y *es un* `MotionListener` ya que la clase implementa *ambas* interfaces. Optamos por implementar ambas interfaces aquí para demostrar una clase que implementa más de una interfaz, pero también pudimos haber implementado la interfaz `MouseInputListener` en su lugar.

Cuando el ratón entra y sale del área de `panelRaton`, se hacen llamadas a los métodos `mouseEntered` (líneas 65 a 71) y `mouseExited` (líneas 74 a 79), respectivamente. El método `mouseEntered` muestra un mensaje en el objeto `barraEstado`, indicando que el ratón entró al objeto `JPanel` y cambia el color de fondo a verde. El método `mouseExited` muestra un mensaje en el objeto `barraEstado`, indicando que el ratón está fuera del objeto `JPanel` (vea la primera ventana de resultados) y cambia el color de fondo a blanco.

Los otros cinco eventos muestran una cadena en el objeto `barraEstado`, la cual contiene el evento y las coordenadas en las que ocurrió. Los métodos **`getX`** y **`getY`** de `MouseEvent` devuelven las coordenadas *x* y *y* del ratón, respectivamente, en el momento en el que ocurrió el evento.

12.15 Clases adaptadoras

Muchas de las interfaces de escucha de eventos, como `MouseListener` y `MouseMotionListener`, contienen varios métodos. No siempre es deseable declarar todos los métodos en una interfaz de escucha de eventos. Por ejemplo, una aplicación podría necesitar solamente el manejador `mouseClicked` de la interfaz `MouseListener`, o el manejador `mouseDragged` de la interfaz `MouseMotionListener`. La interfaz `WindowListener` especifica siete métodos manejadores de eventos de ventana. Para muchas de las interfaces de escucha de eventos que contienen varios métodos, los paquetes `java.awt.event` y `javax.swing.event` proporcionan clases adaptadoras de escucha de eventos. Una **clase adaptadora** implementa a una interfaz y proporciona una implementación predeterminada (con un cuerpo vacío para los métodos) de todos los métodos en la interfaz. En la figura 12.30 se muestran varias clases adaptadoras de `java.awt.event`, junto con las interfaces que implementan. Usted puede extender una clase adaptadora para heredar la implementación predeterminada de cada método, y en consecuencia sobrescribir sólo los métodos que necesite para manejar eventos.

Observación de ingeniería de software 12.6

*Cuando una clase implementa a una interfaz, la clase tiene una relación del tipo "*es un*" con esa interfaz. Todas las subclases directas e indirectas de esa clase heredan esta interfaz. Por lo tanto, un objeto de una clase que extiende a una clase adaptadora de eventos es un objeto del tipo de escucha de eventos correspondiente (por ejemplo, un objeto de una subclase de* `MouseAdapter` *es un* `MouseListener`*).*

Clase adaptadora de eventos en `java.awt.event`	Implementa a la interfaz
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Fig. 12.30 | Las clases adaptadoras de eventos y las interfaces que implementan.

Extensión de `MouseAdapter`

La aplicación de las figuras 12.31 y 12.32 demuestra cómo determinar el número de clics del ratón (es decir, la cuenta de clics) y cómo diferenciar los distintos botones del ratón. El componente de escucha de eventos en esta aplicación es un objeto de la clase interna `ManejadorClicRaton` (figura 12.31, líneas 25 a 46) que extiende a `MouseAdapter`, por lo que podemos declarar sólo el método `mouseClicked` que necesitamos en este ejemplo.

```
1  // Fig. 12.31: MarcoDetallesRaton.java
2  // Demostración de los clics del ratón y cómo diferenciar los botones del mismo.
3  import java.awt.BorderLayout;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8
9  public class MarcoDetallesRaton extends JFrame
10 {
11    private String detalles; // String que se muestra en barraEstado
12    private final JLabel barraEstado; // JLabel que aparece en la parte inferior
                                        de la ventana
13
14    // constructor establece String de la barra de título y registra componente
         de escucha del ratón
15    public MarcoDetallesRaton()
16    {
17       super("Clics y botones del raton");
18
19       barraEstado = new JLabel("Haga clic en el raton");
20       add(barraEstado, BorderLayout.SOUTH);
21       addMouseListener(new ManejadorClicRaton()); // agrega el manejador
22    }
23
24    // clase interna para manejar los eventos del ratón
25    private class ManejadorClicRaton extends MouseAdapter
26    {
27       // maneja evento de clic del ratón y determina cuál botón se oprimió
28       @Override
29       public void mouseClicked(MouseEvent evento)
30       {
```

Fig. 12.31 | Demostración de los clics del ratón y cómo diferenciar los botones del mismo (parte 1 de 2).

```
            int xPos = evento.getX(); // obtiene posición x del ratón
            int yPos = evento.getY(); // obtiene posición y del ratón

            detalles = String.format("Se hizo clic %d vez(veces)",
               evento.getClickCount());

            if (evento.isMetaDown()) // botón derecho del ratón
               detalles += " con el boton derecho del raton";
            else if (evento.isAltDown()) // botón central del ratón
               detalles += " con el boton central del raton";
            else // botón izquierdo del ratón
               detalles += " con el boton izquierdo del raton";

            barraEstado.setText(detalles); // muestra mensaje en barraEstado
         }
      }
   } // fin de la clase MarcoDetallesRaton
```

Fig. 12.31 | Demostración de los clics del ratón y cómo diferenciar los botones del mismo (parte 2 de 2).

```
// Fig. 12.32: DetallesRaton.java
// Prueba de MarcoDetallesRaton.
import javax.swing.JFrame;

public class DetallesRaton
{
   public static void main(String[] args)
   {
      MarcoDetallesRaton marcoDetallesRaton = new MarcoDetallesRaton();
      marcoDetallesRaton.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      marcoDetallesRaton.setSize(400, 150);
      marcoDetallesRaton.setVisible(true);
   }
} // fin de la clase DetallesRaton
```

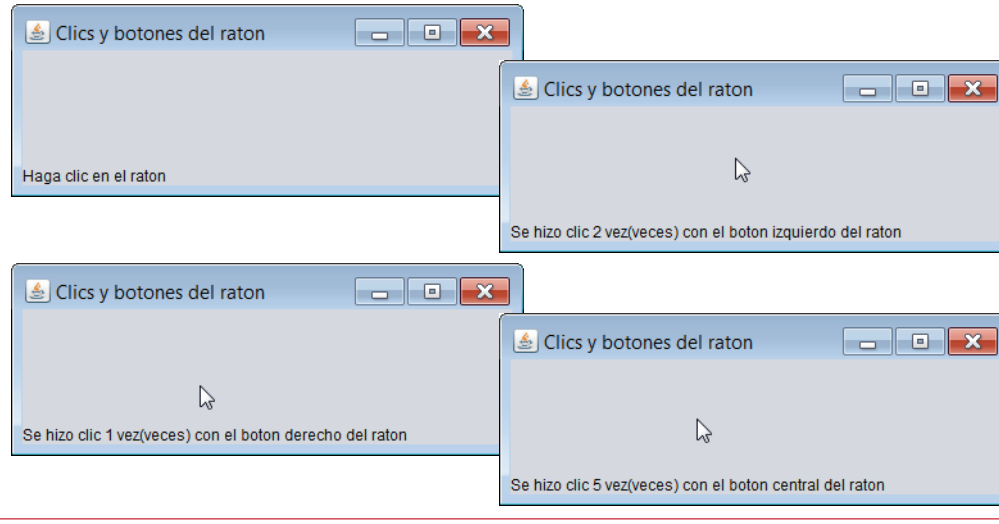


Fig. 12.32 | Prueba de `MarcoDetallesRaton`.

Error común de programación 12.3

Si extiende una clase adaptadora y escribe de manera incorrecta el nombre del método que está sobrescribiendo, y no declara el método con `@Override`*, su método simplemente se vuelve otro método en la clase. Éste es un error lógico difícil de detectar, ya que el programa llamará a la versión vacía del método heredado de la clase adaptadora.*

Un usuario de una aplicación en Java puede estar en un sistema con un ratón de uno, dos o tres botones. Java cuenta con un mecanismo para diferenciar cada uno de los botones del ratón. La clase `MouseEvent` hereda varios métodos de la clase `InputEvent` que pueden diferenciar los botones de un ratón con varios botones, o pueden imitarlo con una combinación de teclas y el clic de un botón del ratón. La figura 12.33 muestra los métodos de `InputEvent` que se utilizan para diferenciar los clics de los botones del ratón. Java asume que cada ratón contiene un botón izquierdo. Por ende, es fácil probar un clic del botón izquierdo del ratón. Sin embargo, los usuarios que tengan un ratón de uno o dos botones deben usar al mismo tiempo una combinación de teclas y clics del ratón, para simular los botones que éste no tiene. En el caso de un ratón con uno o dos botones, una aplicación de Java asume que se hizo clic en el botón central del ratón, si el usuario mantiene oprimida la tecla *Alt* y hace clic en el botón izquierdo en un ratón con dos botones, o en el único botón en un ratón con un botón. En el caso de un ratón con un botón, una aplicación de Java asume que se hizo clic en el botón derecho si el usuario mantiene oprimida la tecla *Meta* (algunas veces conocida como la tecla de *Comando*, o la tecla de la "Manzana" en la Mac) y hace clic en el botón del ratón.

Método `InputEvent`	Descripción
`isMetaDown()`	Devuelve `true` cuando el usuario hace clic en el *botón derecho del ratón*, en un ratón con dos o tres botones. Para simular un clic con el botón derecho del ratón en un ratón con un botón, el usuario puede mantener oprimida la tecla *Meta* en el teclado y hacer clic con el botón del ratón.
`isAltDown()`	Devuelve `true` cuando el usuario hace clic con el *botón central del ratón*, en un ratón con tres botones. Para simular un clic con el botón central del ratón en un ratón con uno o dos botones, el usuario puede oprimir la tecla *Alt* en el teclado y hacer clic en el único botón o en el botón izquierdo del ratón, respectivamente.

Fig. 12.33 | Métodos de `InputEvent` que ayudan a determinar si se hizo clic con el botón derecho o central del ratón.

La línea 21 de la figura 12.31 registra un objeto `MouseListener` para el `MarcoDetallesRaton`. El componente de escucha de eventos es un objeto de la clase `ManejadorClicRaton`, el cual extiende a `MouseAdapter`. Esto nos permite declarar sólo el método `mouseClicked` (líneas 28 a 45). Este método primero captura las coordenadas en donde ocurrió el evento y las almacena en las variables locales `xPos` y `yPos` (líneas 31 y 32). Las líneas 34 y 35 crean un objeto `String` llamado `detalles` que contiene el número de clics del ratón, el cual se devuelve mediante el método `getClickCount` de `MouseEvent` en la línea 35. Las líneas 37 a 42 utilizan los métodos `isMetaDown` e `isAltDown` para determinar cuál botón del ratón oprimió el usuario, y adjuntan un objeto `String` apropiado a `detalles` en cada caso. El objeto `String` resultante se muestra en la `barraEstado`. La clase `DetallesRaton` (figura 12.32) contiene el método `main` que ejecuta la aplicación. Pruebe haciendo clic con cada uno de los botones de su ratón repetidas veces, para ver el incremento en la cuenta de clics.

12.16 Subclase de JPanel para dibujar con el ratón

La sección 12.14 mostró cómo rastrear los eventos del ratón en un objeto `JPanel`. En esta sección usaremos un objeto `JPanel` como un **área dedicada de dibujo**, en la cual el usuario puede dibujar arrastrando el ratón. Además, esta sección demuestra un componente de escucha de eventos que extiende a una clase adaptadora.

Método `paintComponent`

Los componentes ligeros de Swing que extienden a la clase `JComponent` (como `JPanel`) contienen el método **`paintComponent`**, el cual se llama cuando se muestra un componente ligero de Swing. Al sobrescribir este método, puede especificar cómo dibujar figuras usando las herramientas de gráficos de Java. Al personalizar un objeto `JPanel` para usarlo como un área dedicada de dibujo, la subclase debe sobrescribir el método `paintComponent` y llamar a la versión de `paintComponent` de la superclase como la primera instrucción en el cuerpo del método sobrescrito, para asegurar que el componente se muestre en forma correcta. La razón de ello es que las subclases de `JComponent` soportan la **transparencia**. Para mostrar un componente en forma correcta, el programa debe determinar si el componente es transparente. El código que determina esto se encuentra en la implementación del método `paintComponent` de la superclase `JComponent`. Cuando un componente es transparente, `paintComponent` no borra su fondo cuando el programa muestra el componente. Cuando un componente es **opaco**, `paintComponent` borra el fondo del componente antes de mostrarlo. La transparencia de un componente ligero de Swing puede establecerse con el método **`setOpaque`** (un argumento `false` indica que el componente es transparente).

Tip para prevenir errores 12.1

En el método `paintComponent` de una subclase de `JComponent`, la primera instrucción siempre debe ser una llamada al método `paintComponent` de la superclase, para asegurar que un objeto de la subclase se muestre en forma correcta.

Error común de programación 12.4

Si un método `paintComponent` sobrescrito no llama a la versión de la superclase, el componente de la subclase tal vez no se muestre en forma apropiada. Si un método `paintComponent` sobrescrito llama a la versión de la superclase después de realizar otro dibujo, éste se borra.

Definición del área de dibujo personalizada

La aplicación `Pintor` de las figuras 12.34 y 12.35 demuestra una subclase personalizada de `JPanel` que se utiliza para crear un área dedicada de dibujo. La aplicación utiliza el manejador de eventos `mouseDragged` para crear una aplicación simple de dibujo. El usuario puede dibujar imágenes arrastrando el ratón en el objeto `JPanel`. Este ejemplo no utiliza el método `mouseMoved`, por lo que nuestra *clase de escucha de eventos* (la *clase interna anónima* en las líneas 20 a 29) extiende a `MouseMotionAdapter`. Como esta clase ya declara tanto a `mouseMoved` como `mouseDragged`, simplemente podemos sobrescribir a `mouseDragged` para proporcionar el manejo de eventos que requiere esta aplicación.

```
1  // Fig. 12.34: PanelDibujo.java
2  // Clase adaptadora que se usa para implementar manejadores de eventos.
3  import java.awt.Point;
4  import java.awt.Graphics;
5  import java.awt.event.MouseEvent;
```

Fig. 12.34 | Clase adaptadora que se usa para implementar los manejadores de eventos (parte 1 de 2).

```
6  import java.awt.event.MouseMotionAdapter;
7  import java.util.ArrayList;
8  import javax.swing.JPanel;
9
10 public class PanelDibujo extends JPanel
11 {
12    // lista de referencias Point
13    private final ArrayList<Point> puntos = new ArrayList<>();
14
15    // establece la GUI y registra el manejador de eventos del ratón
16    public PanelDibujo()
17    {
18       // maneja evento de movimiento del ratón en el marco
19       addMouseMotionListener(
20          new MouseMotionAdapter() // clase interna anónima
21          {
22             // almacena las coordenadas de arrastre y vuelve a dibujar
23             @Override
24             public void mouseDragged(MouseEvent evento)
25             {
26                puntos.add(evento.getPoint());
27                repaint(); // vuelve a dibujar JFrame
28             }
29          }
30       );
31    }
32
33    // dibuja óvalos en un cuadro delimitador de 4 x 4, en las ubicaciones
         especificadas en la ventana
34    @Override
35    public void paintComponent(Graphics g)
36    {
37       super.paintComponent(g); // borra el área de dibujo
38
39       // dibuja todos los puntos
40       for (Point punto : puntos)
41          g.fillOval(punto.x, punto.y, 4, 4);
42    }
43 } // fin de la clase PanelDibujo
```

Fig. 12.34 | Clase adaptadora que se usa para implementar los manejadores de eventos (parte 2 de 2).

La clase `PanelDibujo` (figura 12.34) extiende a `JPanel` para crear el área dedicada de dibujo. La clase **`Point`** (paquete `java.awt`) representa una coordenada *x*-*y*. Utilizamos objetos de esta clase para almacenar las coordenadas de cada evento de arrastre del ratón. La clase **`Graphics`** se utiliza para dibujar. En este ejemplo, utilizamos un objeto `ArrayList` de objetos `Point` (línea 13) para almacenar la ubicación en la cual ocurre cada evento de arrastre del ratón. Como veremos más adelante, el método `paintComponent` utiliza estos objetos `Point` para dibujar.

Las líneas 19 a 30 registran un objeto `MouseMotionListener` para que escuche los eventos de movimiento del ratón de `PaintPanel`. Las líneas 20 a 29 crean un objeto de una clase interna anónima que extiende a la clase adaptadora `MouseMotionAdapter`. Recuerde que `MouseMotionAdapter` implementa a `MouseMotionListener`, por lo que el objeto de la *clase interna anónima* es un `MouseMotionListener`. La clase interna anónima hereda una implementación predeterminada de los métodos `mouseMoved` y `mouseDragged`, por lo que de antemano implementa a todos los métodos de la interfaz. Sin embargo, las

implementaciones predeterminadas no hacen nada cuando se les llama. Por lo tanto, sobrescribimos el método `mouseDragged` en las líneas 23 a 28 para capturar las coordenadas de un evento de arrastre del ratón y las almacenamos como un objeto `Point`. La línea 26 invoca el método **`getPoint`** de `MouseEvent` para obtener el objeto `Point` en donde ocurrió el evento, y lo almacena en el objeto `ArrayList`. La línea 27 llama al método **`repaint`** (heredado directamente de la clase `Component`) para indicar que el objeto `PanelDibujo` debe actualizarse en la pantalla lo más pronto posible, con una llamada al método `paintComponent` de `PaintPanel`.

El método `paintComponent` (líneas 34 a 42), que recibe un parámetro `Graphics`, se llama de manera automática cada vez que el objeto `PaintPanel` necesita mostrarse en la pantalla (como cuando se muestra por primera vez la GUI) o actualizarse en la pantalla (como cuando se hace una llamada al método `repaint`, o cuando otra ventana en la pantalla *oculta* el componente de la GUI y después se vuelve otra vez visible).

Observación de apariencia visual 12.13

Una llamada a `repaint` para un componente de GUI de Swing indica que el componente debe actualizarse en la pantalla lo más pronto posible. El fondo del componente se borra sólo si el componente es opaco. El método `setOpaque` de `JComponent` puede recibir un argumento `boolean`, el cual indica si el componente es opaco (`true`) o transparente(`false`).

La línea 37 invoca a la versión de `paintComponent` de la superclase para borrar el fondo de `PanelDibujo` (los objetos `JPanel` son opacos de manera predeterminada). Las líneas 40 y 41 dibujan un óvalo en la ubicación especificada por cada objeto `Point` en el arreglo `ArrayList`. El método **`fillOval`** de `Graphics` dibuja un óvalo relleno. Los cuatro parámetros del método representan un área rectangular (que se conoce como *cuadro delimitador*) en la cual se muestra el óvalo. Los primeros dos parámetros son la coordenada *x* superior izquierda y la coordenada *y* superior izquierda del área rectangular. Las últimas dos coordenadas representan la anchura y la altura del área rectangular. El método `fillOval` dibuja el óvalo de manera que esté en contacto con la parte media de cada lado del área rectangular. En la línea 41, los primeros dos argumentos se especifican mediante el uso de las dos variables de instancia `public` de la clase `Point`: `x` y `y`. En el capítulo 13 aprenderá sobre más características de `Graphics`.

Observación de apariencia visual 12.14

La acción de dibujar en cualquier componente de GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0,0) de ese componente de la GUI, no de la esquina superior izquierda de la pantalla.

Uso del objeto `JPanel` personalizado en una aplicación

La clase `Pintor` (figura 12.35) contiene el método `main` que ejecuta esta aplicación. En la línea 14 se crea un objeto `PanelDibujo`, en el cual el usuario puede arrastrar el ratón para dibujar. En la línea 15 se adjunta el objeto `PanelDibujo` al objeto `JFrame`.

```
1  // Fig. 12.35: Pintor.java
2  // Prueba de PanelDibujo.
3  import java.awt.BorderLayout;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6
7  public class Pintor
8  {
```

Fig. 12.35 | Prueba de `PanelDibujo` (parte 1 de 2).

```
9     public static void main(String[] args)
10    {
11       // crea objeto JFrame
12       JFrame aplicacion = new JFrame("Un programa simple de dibujo");
13
14       PanelDibujo panelDibujo = new PanelDibujo();
15       aplicacion.add(panelDibujo, BorderLayout.CENTER);
16
17       // crea una etiqueta y la coloca en la región SOUTH de BorderLayout
18       aplicacion.add(new JLabel("Arrastre el raton para dibujar"),
19          BorderLayout.SOUTH);
20
21       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22       aplicacion.setSize(400, 200);
23       aplicacion.setVisible(true);
24    }
25 } // fin de la clase Pintor
```

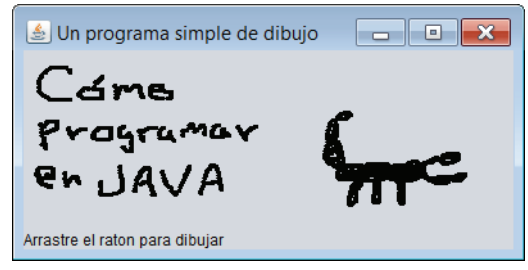


Fig. 12.35 | Prueba de `PanelDibujo` (parte 2 de 2).

12.17 Manejo de eventos de teclas

En esta sección presentamos la interfaz `KeyListener` para manejar **eventos de teclas**. Estos eventos se generan cuando se oprimen y sueltan las teclas en el teclado. Una clase que implementa a `KeyListener` debe proporcionar declaraciones para los métodos **`keyPressed`**, **`keyReleased`** y **`keyTyped`**, cada uno de los cuales recibe un objeto `KeyEvent` como argumento. La clase `KeyEvent` es una subclase de `InputEvent`. El método `keyPressed` es llamado en respuesta a la acción de oprimir cualquier tecla. El método `keyTyped` es llamado en respuesta a la acción de oprimir una tecla que no sea una **tecla de acción**. (Las teclas de acción son cualquier tecla de dirección, *Inicio*, *Fin*, *Re Pág*, *Av Pág*, cualquier tecla de función, etc.). El método `keyReleased` es llamado cuando la tecla se suelta después de un evento `keyPressed` o `keyTyped`.

La aplicación de las figuras 12.36 y 12.37 demuestra el uso de los métodos de `KeyListener`. La clase `MarcoDemoTeclas` implementa la interfaz `KeyListener`, por lo que los tres métodos se declaran en la aplicación. El constructor (figuras 12.36, líneas 17 a 28) registra a la aplicación para manejar sus propios eventos de teclas, utilizando el método **`addKeyListener`** en la línea 27. Este método se declara en la clase `Component`, por lo que todas las subclases de `Component` pueden notificar a objetos `KeyListener` acerca de los eventos de teclas para ese objeto `Component`.

```
1  // Fig. 12.36: MarcoDemoTeclas.java
2  // Manejo de eventos de teclas.
3  import java.awt.Color;
```

Fig. 12.36 | Manejo de eventos de teclas (parte 1 de 3).

```
4  import java.awt.event.KeyListener;
5  import java.awt.event.KeyEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8
9  public class MarcoDemoTeclas extends JFrame implements KeyListener
10 {
11    private String linea1 = ""; // primera línea del área de texto
12    private String linea2 = ""; // segunda línea del área de texto
13    private String linea3 = ""; // tercera línea del área de texto
14    private JTextArea areaTexto; // área de texto para mostrar la salida
15
16    // constructor de MarcoDemoTeclas
17    public MarcoDemoTeclas()
18    {
19       super("Demostracion de los eventos de pulsacion de teclas");
20
21       areaTexto = new JTextArea(10, 15); // establece el objeto JTextArea
22       areaTexto.setText("Oprima cualquier tecla en el teclado...");
23       areaTexto.setEnabled(false);
24       areaTexto.setDisabledTextColor(Color.BLACK);
25       add(areaTexto); // agrega el área de texto a JFrame
26
27       addKeyListener(this); // permite al marco procesar los eventos de teclas
28    }
29
30    // maneja el evento de oprimir cualquier tecla
31    @Override
32    public void keyPressed(KeyEvent evento)
33    {
34       linea1 = String.format("Tecla oprimida: %s",
35          KeyEvent.getKeyText(evento.getKeyCode())); // muestra la tecla oprimida
36       establecerLineas2y3(evento); // establece las líneas de salida dos y tres
37    }
38
39    // maneja el evento de liberar cualquier tecla
40    @Override
41    public void keyReleased(KeyEvent evento)
42    {
43       linea1 = String.format("Tecla liberada: %s",
44          KeyEvent.getKeyText(evento.getKeyCode())); // muestra la tecla liberada
45       establecerLineas2y3(evento); // establece las líneas de salida dos y tres
46    }
47
48    // maneja el evento de oprimir una tecla de acción
49    @Override
50    public void keyTyped(KeyEvent evento)
51    {
52       linea1 = String.format("Tecla oprimida: %s", evento.getKeyChar());
53       establecerLineas2y3(evento); // establece las líneas de salida dos y tres
54    }
55
```

Fig. 12.36 | Manejo de eventos de teclas (parte 2 de 3).

```
56     // establece las líneas de salida dos y tres
57     private void establecerLineas2y3(KeyEvent evento)
58     {
59        linea2 = String.format("Esta tecla %s es una tecla de accion",
60           (evento.isActionKey() ? "" : "no "));
61
62        String temp = KeyEvent.getKeyModifiersText(evento.getModifiers());
63
64        linea3 = String.format("Teclas modificadoras oprimidas: %s",
65           (temp.equals("") ? "ninguna" : temp)); // imprime modificadoras
66
67        areaTexto.setText(String.format("%s\n%s\n%s\n",
68           linea1, linea2, linea3)); // imprime tres líneas de texto
69     }
70  } // fin de la clase MarcoDemoTeclas
```

Fig. 12.36 | Manejo de eventos de teclas (parte 3 de 3).

```
1  // Fig. 12.37: DemoTeclas.java
2  // Prueba de MarcoDemoTeclas.
3  import javax.swing.JFrame;
4
5  public class DemoTeclas
6  {
7     public static void main(String[] args)
8     {
9        MarcoDemoTeclas marcoDemoTeclas = new MarcoDemoTeclas();
10       marcoDemoTeclas.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoDemoTeclas.setSize(350, 100);
12       marcoDemoTeclas.setVisible(true);
13    }
14 } // fin de la clase DemoTeclas
```

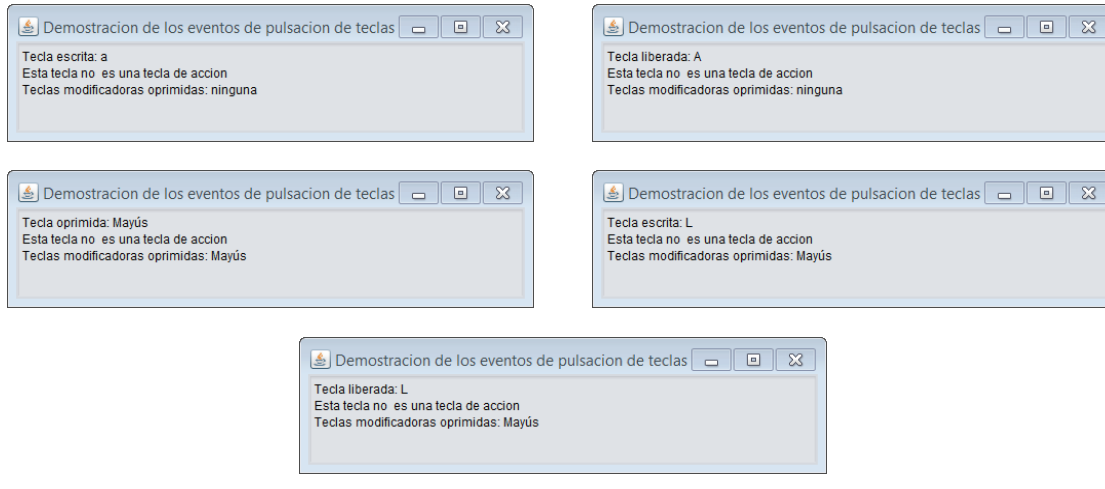


Fig. 12.37 | Prueba de `MarcoDemoTeclas` (parte 1 de 2).

Fig. 12.37 | Prueba de `MarcoDemoTeclas` (parte 2 de 2).

En la línea 25, el constructor agrega el objeto `JTextArea` llamado `areaTexto` (en donde se muestra la salida de la aplicación) al objeto `JFrame`. Un objeto `JTextArea` es un área multilínea en la que podemos mostrar texto (hablaremos sobre los objetos `JTextArea` con más detalle en la sección 12.20). Observe en las capturas de pantalla que el objeto `areaTexto` ocupa *toda la ventana*. Esto se debe al esquema predeterminado `BorderLayout` del objeto `JFrame` (que describiremos en la sección 12.18.2 y demostraremos en la figura 12.41). Cuando se agrega un objeto `Component` individual a un objeto `BorderLayout`, el objeto `Component` ocupa *todo* el objeto `Container`. La línea 23 deshabilita el objeto `JTextArea` para que el usuario no pueda escribir en él. Esto hace que el objeto `JTextArea` se torne de color gris. En la línea 24 se utiliza el método **`setDisabledTextColor`** para cambiar el color del texto en el área de texto `JTextArea` a negro para mejorar la legibilidad.

Los métodos `keyPressed` (líneas 31 a 37) y `keyReleased` (líneas 40 a 46) utilizan el método **`getKeyCode`** de `KeyEvent` para obtener el **código de tecla virtual** de la tecla oprimida. La clase `KeyEvent` mantiene un conjunto de constantes de códigos virtuales que representan a todas las teclas en el teclado. Estas constantes pueden compararse con el valor de retorno de `getKeyCode` para probar teclas individuales en el teclado. El valor devuelto por `getKeyCode` se pasa al método **`getKeyText`** de `KeyEvent`, el cual devuelve una cadena que contiene el nombre de la tecla que se oprimió. Para obtener una lista completa de las constantes de teclas virtuales, vea la documentación en línea para la clase `KeyEvent` (paquete `java.awt.event`). El método `keyTyped` (líneas 49 a 54) utiliza el método **`getKeyChar`** de `KeyEvent` (el cual devuelve un valor `char`) para obtener el valor Unicode del carácter escrito.

Los tres métodos manejadores de eventos terminan llamando al método `establecerLineas2y3` (líneas 57 a 69) y le pasan el objeto `KeyEvent`. Este método utiliza el método **`isActionKey`** de `KeyEvent` (línea 60) para determinar si la tecla en el evento fue una tecla de acción. Además, se hace una llamada al método **`getModifiers`** de `InputEvent` (línea 62) para determinar si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla. El resultado de este método se pasa al método **`getKeyModifiersText`** de `KeyEvent`, el cual produce un objeto `String` que contiene los nombres de las teclas modificadoras que se oprimieron.

[*Nota*: si necesita probar una tecla específica en el teclado, la clase `KeyEvent` proporciona una **constante de tecla** para cada tecla del teclado. Estas constantes pueden utilizarse desde los manejadores de eventos de teclas para determinar si se oprimió una tecla específica. Además, para determinar si las teclas *Alt*, *Ctrl*, *Meta* y *Mayús* se oprimen individualmente, cada uno de los métodos **`isAltDown`**, **`isControlDown`**, **`isMetaDown`** e **`isShiftDown`** devuelven un valor `boolean`, indicando si se oprimió dicha tecla durante el evento de tecla].

12.18 Introducción a los administradores de esquemas

Los **administradores de esquemas** *ordenan* los componentes de la GUI en un contenedor, para fines de presentación. Los programadores pueden usar los administradores de esquemas como herramientas básicas de distribución visual, en vez de determinar la posición y tamaño exactos de cada componente de la GUI. Esta funcionalidad permite al programador concentrarse en la apariencia general, y deja que el administrador de esquemas procese la mayoría de los detalles de la distribución visual. Todos los administradores de esquemas implementan la interfaz **`LayoutManager`** (en el paquete `java.awt`). El método `setLayout` de la

clase `Container` toma un objeto que implementa a la interfaz `LayoutManager` como argumento. Básicamente, existen tres formas para poder ordenar los componentes en una GUI:

1. *Posicionamiento absoluto*: esto proporciona el mayor nivel de control sobre la apariencia de una GUI. Al establecer el esquema de un objeto `Container` en `null`, podemos especificar la *posición absoluta de cada componente de la GUI* con respecto a la esquina superior izquierda del objeto `Container`, usando los métodos `setSize` y `setLocation` o `setBounds` de `Component`. Si hacemos esto, también debemos especificar el tamaño de cada componente de la GUI. La programación de una GUI con posicionamiento absoluto puede ser un proceso tedioso, a menos que se cuente con un entorno de desarrollo integrado (IDE), que pueda generar el código por nosotros.
2. *Administradores de esquemas*: el uso de administradores de esquemas para posicionar elementos puede ser un proceso más simple y rápido que la creación de una GUI con posicionamiento absoluto, pero se pierde cierto control sobre el tamaño y el posicionamiento preciso de los componentes de la GUI.
3. *Programación visual en un IDE*: los IDE proporcionan herramientas que facilitan la creación de GUI. Por lo general, cada IDE proporciona una **herramienta de diseño de GUI** que nos permite arrastrar y soltar componentes de GUI desde un cuadro de herramientas hacia un área de diseño. Después podemos posicionar, ajustar el tamaño de los componentes de la GUI y alinearlos según lo deseado. El IDE genera el código de Java que crea la GUI. Además, por lo general podemos agregar código manejador de eventos para un componente específico, haciendo doble clic en el componente. Algunas herramientas de diseño también nos permiten utilizar los administradores de esquemas descritos en este capítulo y en el capítulo 22.

Observación de apariencia visual 12.15

La mayoría de los IDE de Java proporcionan herramientas de diseño de GUI para crear una GUI en forma visual; posteriormente, las herramientas escriben código en Java para crear la GUI. Dichas herramientas a menudo proporcionan un mayor control sobre el tamaño, la posición y la alineación de los componentes de la GUI, en comparación con los administradores de esquemas integrados.

Observación de apariencia visual 12.16

Es posible establecer el esquema de un objeto `Container` en `null`, lo cual indica que no debe utilizarse ningún administrador de esquemas. En un objeto `Container` sin un administrador de esquemas, el programador debe posicionar y cambiar el tamaño de los componentes en el contenedor dado, y cuidar que, en los eventos de ajuste de tamaño, todos los componentes se reposicionen según sea necesario. Los eventos de ajuste de tamaño de un componente pueden procesarse mediante un objeto `ComponentListener`.

En la figura 12.38 se sintetizan los administradores de esquemas presentados en este capítulo. En el capítulo 22 en línea hablaremos sobre un par de administradores de esquemas adicionales.

Administrador de esquemas	Descripción
`FlowLayout`	Es el predeterminado para `javax.swing.JPanel`. Coloca los componentes *secuencialmente* (*de izquierda a derecha*) en el orden en que se agregaron. También es posible especificar el orden de los componentes utilizando el método `add` de `Container`, el cual toma un objeto `Component` y una posición de índice entero como argumentos.
`BorderLayout`	Es el predeterminado para los objetos `JFrame` (y otras ventanas). Ordena los componentes en cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`.
`GridLayout`	Ordena los componentes en filas y columnas

Fig. 12.38 | Administradores de esquemas.

12.18.1 FlowLayout

Éste es el administrador de esquemas *más simple*. Los componentes de la GUI se colocan en un contenedor, de izquierda a derecha, en el orden en el que se agregaron. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. La clase `FlowLayout` permite a los componentes de la GUI *alinearse a la izquierda*, *al centro* (el valor predeterminado) y *a la derecha*.

La aplicación de las figuras 12.39 y 12.40 crea tres objetos `JButton` y los agrega a la aplicación, utilizando un administrador de esquemas `FlowLayout`. Los componentes se alinean hacia el centro de manera predeterminada. Cuando el usuario hace clic en **Izquierda**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la izquierda. Cuando el usuario hace clic en **Derecha**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la derecha. Cuando el usuario hace clic en **Centro**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado hacia el centro. Las ventanas de salida de ejemplo muestran cada una de las alineaciones. La última ventana de salida de ejemplo muestra la alineación centrada después de ajustar el tamaño de la ventana a una anchura menor, de modo que el botón **Derecha** fluya hacia una nueva línea.

Como se vio antes, el esquema de un contenedor se establece mediante el método `setLayout` de la clase `Container`. En la línea 25 (figura 12.39) se establece el administrador de esquemas en `FlowLayout`, el cual se declara en la línea 23. Por lo general, el esquema se establece antes de agregar cualquier componente de la GUI a un contenedor.

Observación de apariencia visual 12.17

Cada contenedor individual puede tener solamente un administrador de esquemas, pero varios contenedores en la misma aplicación pueden tener distintos administradores de esquemas.

```
1  // Fig. 12.39: MarcoFlowLayout.java
2  // FlowLayout permite que los componentes fluyan a través de varias líneas.
3  import java.awt.FlowLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class MarcoFlowLayout extends JFrame
11 {
12    private final JButton botonJButtonIzquierda; // botón para establecer la
                                                      alineación a la izquierda
13    private final JButton botonJButtonCentro; // botón para establecer la
                                                 alineación al centro
14    private final JButton botonJButtonDerecha; // botón para establecer la
                                                  alineación a la derecha
15    private final FlowLayout esquema; // objeto esquema
16    private final Container contenedor; // contenedor para establecer el esquema
17
18    // establece la GUI y registra los componentes de escucha de botones
19    public MarcoFlowLayout()
20    {
21       super("Demostracion de FlowLayout");
22
23       esquema = new FlowLayout();
24       contenedor = getContentPane(); // obtiene contenedor para esquema
25       setLayout(esquema);
26
```

Fig. 12.39 | `FlowLayout` permite que los componentes fluyan a través de varias líneas (parte 1 de 2).

```
27         // establece botonJButtonIzquierda y registra componente de escucha
28         botonJButtonIzquierda = new JButton("Izquierda");
29         add(botonJButtonIzquierda); // agrega botón Izquierda al marco
30         botonJButtonIzquierda.addActionListener(
31            new ActionListener() // clase interna anónima
32            {
33               // procesa evento de botonJButtonIzquierda
34               @Override
35               public void actionPerformed(ActionEvent evento)
36               {
37                  esquema.setAlignment(FlowLayout.LEFT);
38
39                  // realinea los componentes adjuntos
40                  esquema.layoutContainer(contenedor);
41               }
42            }
43         );
44
45         // establece botonJButtonCentro y registra componente de escucha
46         botonJButtonCentro = new JButton("Centro");
47         add(botonJButtonCentro); // agrega botón Centro al marco
48         botonJButtonCentro.addActionListener(
49            new ActionListener() // clase interna anónima
50            {
51               // procesa evento de botonJButtonCentro
52               @Override
53               public void actionPerformed(ActionEvent evento)
54               {
55                  esquema.setAlignment(FlowLayout.CENTER);
56
57                  // realinea los componentes adjuntos
58                  esquema.layoutContainer(contenedor);
59               }
60            }
61         );
62
63         // establece botonJButtonDerecha y registra componente de escucha
64         botonJButtonDerecha = new JButton("Derecha");
65         add(botonJButtonDerecha); // agrega botón Derecha al marco
66         botonJButtonDerecha.addActionListener(
67            new ActionListener() // clase interna anónima
68            {
69               // procesa evento de botonJButtonDerecha
70               @Override
71               public void actionPerformed(ActionEvent evento)
72               {
73                  esquema.setAlignment(FlowLayout.RIGHT);
74
75                  // realinea los componentes adjuntos
76                  esquema.layoutContainer(contenedor);
77               }
78            }
79         );
80      } // fin del constructor de MarcoFlowLayout
81   } // fin de la clase MarcoFlowLayout
```

Fig. 12.39 | `FlowLayout` permite que los componentes fluyan a través de varias líneas (parte 2 de 2).

```
1  // Fig. 12.40: DemoFlowLayout.java
2  // Prueba MarcoFlowLayout.
3  import javax.swing.JFrame;
4
5  public class DemoFlowLayout
6  {
7     public static void main(String[] args)
8     {
9        MarcoFlowLayout marcoFlowLayout = new MarcoFlowLayout();
10       marcoFlowLayout.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoFlowLayout.setSize(300, 75);
12       marcoFlowLayout.setVisible(true);
13    }
14 } // fin de la clase DemoFlowLayout
```

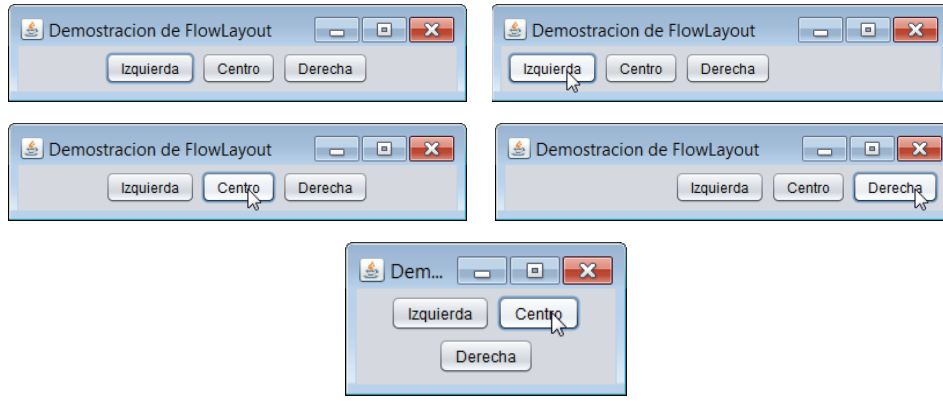


Fig. 12.40 | Prueba `MarcoFlowLayout`.

El manejador de eventos de cada botón se especifica con un objeto de una clase interna anónima independiente (líneas 30 a 43, 48 a 61 y 66 a 79, respectivamente) y el método `actionPerformed` en cada caso ejecuta dos instrucciones. Por ejemplo, la línea 37 en el manejador de eventos para el botón `botonJButtonIzquierda` utiliza el método **`setAlignment`** de `FlowLayout` para cambiar la alineación del objeto `FlowLayout` a la izquierda (**`FlowLayout.LEFT`**). En la línea 40 se utiliza el método **`layoutContainer`** de la interfaz `LayoutManager` (que todos los administradores de esquemas heredan) para especificar que el objeto `JFrame` debe reordenarse, con base en el esquema ajustado. Dependiendo del botón oprimido, el método `actionPerformed` para cada botón establece la alineación del objeto `FlowLayout` a `FlowLayout.LEFT` (línea 37), **`FlowLayout.CENTER`** (línea 55) o **`FlowLayout.RIGHT`** (línea 73).

12.18.2 BorderLayout

El administrador de esquemas `BorderLayout` (el predeterminado para un objeto `JFrame`) ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. `NORTH` corresponde a la parte superior del contenedor. La clase `BorderLayout` extiende a `Object` e implementa a la interfaz **`LayoutManager2`** (una subinterfaz de `LayoutManager`, que agrega varios métodos para un mejor procesamiento de los esquemas).

Un `BorderLayout` limita a un objeto `Container` para que contenga *cuando mucho cinco componentes*; uno en cada región. El componente que se coloca en cada región puede ser un contenedor, al cual se pueden adjuntar otros componentes. Los componentes que se colocan en las regiones `NORTH` y `SOUTH` se extienden horizontalmente hacia los lados del contenedor, y tienen la misma altura que los componentes

que se colocan en esas regiones. Las regiones EAST y WEST se expanden verticalmente entre las regiones NORTH y SOUTH, y tienen la misma anchura que los componentes que se coloquen dentro de ellas. El componente que se coloca en la región CENTER se *expande para rellenar todo el espacio restante en el esquema* (esto explica por qué el objeto JTextArea de la figura 12.37 ocupa toda la ventana). Si las cinco regiones están ocupadas, todo el espacio del contenedor se cubre con los componentes de la GUI. Si las regiones NORTH o SOUTH no están ocupadas, los componentes de la GUI en las regiones EAST, CENTER y WEST se expanden verticalmente para rellenar el espacio restante. Si las regiones EAST o WEST no están ocupadas, el componente de la GUI en la región CENTER *se expande horizontalmente para rellenar el espacio restante*. Si la región CENTER *no* está ocupada, el área se deja *vacía*; los demás componentes de la GUI *no* se expanden para rellenar el espacio restante. La aplicación de las figuras 12.41 y 12.42 demuestra el administrador de esquemas BorderLayout mediante el uso de cinco objetos JButton.

```
1  // Fig. 12.41: MarcoBorderLayout.java
2  // BorderLayout que contiene cinco botones.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class MarcoBorderLayout extends JFrame implements ActionListener
10 {
11    private final JButton botones[]; // arreglo de botones para ocultar porciones
12    private static final String nombres[] = {"Ocultar Norte", "Ocultar Sur",
13       "Ocultar Este", "Ocultar Oeste", "Ocultar Centro"};
14    private final BorderLayout esquema;
15
16    // establece la GUI y el manejo de eventos
17    public MarcoBorderLayout()
18    {
19       super("Demostracion de BorderLayout");
20
21       esquema = new BorderLayout(5, 5); // espacios de 5 píxeles
22       setLayout(esquema);
23       botones = new JButton[nombres.length];
24
25       // crea objetos JButton y registra componentes de escucha para ellos
26       for (int cuenta = 0; cuenta < nombres.length; cuenta++)
27       {
28          botones[cuenta] = new JButton(nombres[cuenta]);
29          botones[cuenta].addActionListener(this);
30       }
31
32       add(botones[0], BorderLayout.NORTH);
33       add(botones[1], BorderLayout.SOUTH);
34       add(botones[2], BorderLayout.EAST);
35       add(botones[3], BorderLayout.WEST);
36       add(botones[4], BorderLayout.CENTER);
37    }
38
```

Fig. 12.41 | BorderLayout que contiene cinco botones (parte 1 de 2).

```
39     // maneja los eventos de botón
40     @Override
41     public void actionPerformed(ActionEvent evento)
42     {
43        // comprueba el origen del evento y distribuye el panel de contenido de
             manera acorde
44        for (JButton boton : botones)
45        {
46           if (evento.getSource() == boton)
47              boton.setVisible(false); // oculta el botón oprimido
48           else
49              boton.setVisible(true); // muestra los demás botones
50        }
51
52        esquema.layoutContainer(getContentPane()); // distribuye el panel de contenido
53     }
54  } // fin de la clase MarcoBorderLayout
```

Fig. 12.41 | `BorderLayout` que contiene cinco botones (parte 2 de 2).

En la línea 21 de la figura 12.41 se crea un objeto `BorderLayout`. Los argumentos del constructor especifican el número de píxeles entre los componentes que se ordenan en forma horizontal (**espacio libre horizontal**) y entre los componentes que se ordenan en forma vertical (**espacio libre vertical**), respectivamente. El valor predeterminado es un píxel de espacio libre horizontal y vertical. En la línea 22 se utiliza el método `setLayout` para establecer el esquema del panel de contenido en `esquema`.

Agregamos objetos `Component` a un objeto `BorderLayout` con otra versión del método `add` de `Container` que toma dos argumentos: el objeto `Component` que se va a agregar y la región en la que debe aparecer este objeto. Por ejemplo, en la línea 32 se especifica que `botones[0]` debe aparecer en la región `NORTH`. Los componentes pueden agregarse en *cualquier* orden, pero sólo debe agregarse *un* componente a cada región.

Observación de apariencia visual 12.18

Si no se especifica una región al agregar un objeto `Component` a un objeto `BorderLayout`, el administrador de esquemas asume que el objeto `Component` debe agregarse a la región `BorderLayout.CENTER`.

Error común de programación 12.5

Cuando se agrega más de un componente a una región en un objeto `BorderLayout`, sólo se mostrará el último componente agregado a esa región. No hay un error que indique este problema.

La clase `MarcoBorderLayout` implementa directamente a `ActionListener` en este ejemplo, por lo que el objeto `MarcoBorderLayout` manejará los eventos de los objetos `JButton`. Por esta razón, en la línea 29 se pasa la referencia `this` al método `addActionListener` de cada objeto `JButton`. Cuando el usuario hace clic en un objeto `JButton` específico en el esquema, se ejecuta el método `actionPerformed` (líneas 40 a 53). La instrucción `for` mejorada en las líneas 44 a 50 utiliza una instrucción `if...else` para ocultar el objeto `JButton` específico que generó el evento. El método **`setVisible`** (que `JButton` hereda de la clase `Component`) se llama con un argumento `false` (línea 47) para ocultar el objeto `JButton`. Si el objeto `JButton` actual en el arreglo no es el que generó el evento, se hace una llamada al método `setVisible` con un argumento `true` (línea 49) para asegurar que el objeto `JButton` se muestre en la pantalla.

En la línea 52 se utiliza el método `layoutContainer` de `LayoutManager` para recalcular la distribución visual del panel de contenido. Observe en las capturas de pantalla de la figura 12.42 que ciertas regiones en el objeto `BorderLayout` cambian de forma a medida que se *ocultan* objetos `JButton` y se muestran en otras regiones. Pruebe a cambiar el tamaño de la ventana de la aplicación para ver cómo las diversas regiones ajustan su tamaño con base en la anchura y la altura de la ventana. *Para esquemas más complejos, agrupe los componentes en objetos `JPanel`, cada uno con un administrador de esquemas separado*. Coloque los objetos `JPanel` en el objeto `JFrame`, usando el esquema `BorderLayout` predeterminado o cualquier otro esquema.

```
1  // Fig. 12.42: DemoBorderLayout.java
2  // Prueba de MarcoBorderLayout.
3  import javax.swing.JFrame;
4
5  public class DemoBorderLayout
6  {
7     public static void main(String[] args)
8     {
9        MarcoBorderLayout marcoBorderLayout = new MarcoBorderLayout();
10       marcoBorderLayout.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoBorderLayout.setSize(300, 200);
12       marcoBorderLayout.setVisible(true);
13    }
14 } // fin de la clase DemoBorderLayout
```

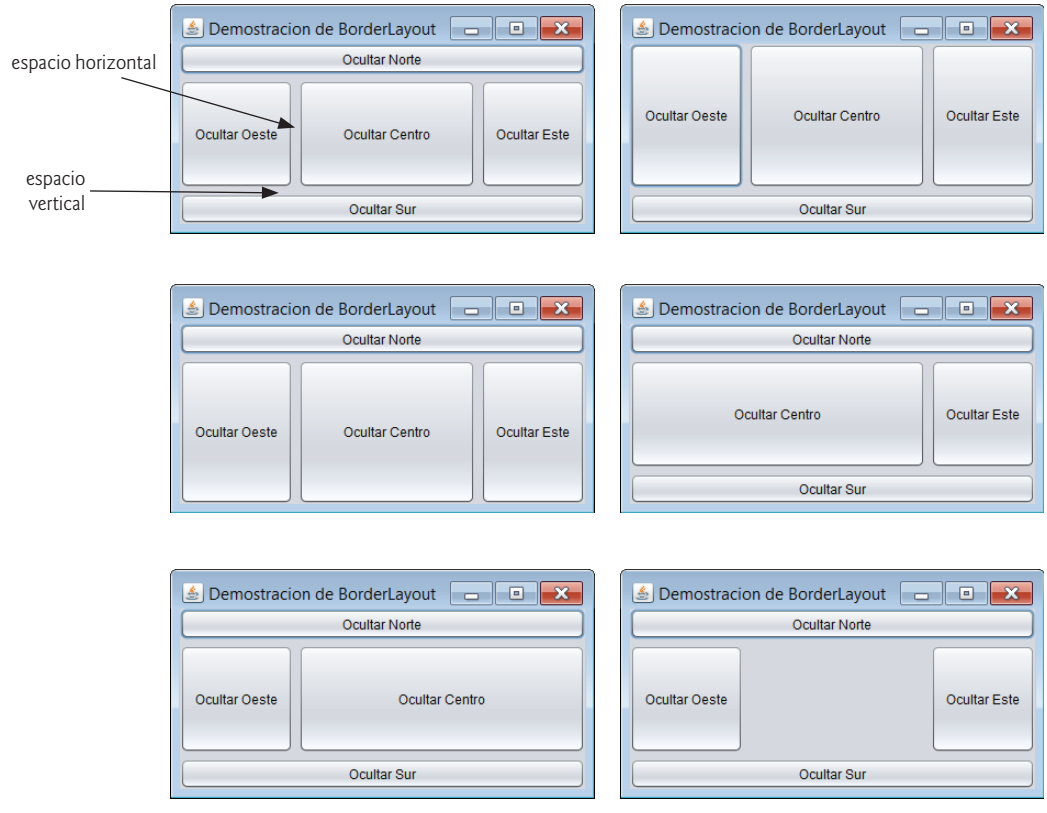


Fig. 12.42 | Prueba de `MarcoBorderLayout`.

12.18.3 GridLayout

El administrador de esquemas **GridLayout** divide el contenedor en *una cuadrícula*, de manera que los componentes puedan colocarse en *filas* y *columnas*. La clase GridLayout hereda directamente de la clase Object e implementa a la interfaz LayoutManager. Todo objeto Component en un objeto GridLayout tiene la *misma* anchura y altura. Los componentes se agregan a un objeto GridLayout empezando en la celda superior izquierda de la cuadrícula, y procediendo de izquierda a derecha hasta que la fila esté llena. Después el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente. La aplicación de las figuras 12.43 y 12.44 demuestra el administrador de esquemas GridLayout, utilizando seis objetos JButton.

```
1  // Fig. 12.43: MarcoGridLayout.java
2  // GridLayout que contiene seis botones.
3  import java.awt.GridLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class MarcoGridLayout extends JFrame implements ActionListener
11 {
12    private final JButton[] botones; // arreglo de botones
13    private static final String[] nombres =
14       { "uno", "dos", "tres", "cuatro", "cinco", "seis" };
15    private boolean alternar = true; // alterna entre dos esquemas
16    private Container contenedor; // contenedor del marco
17    private GridLayout cuadricula1; // primer objeto GridLayout
18    private GridLayout cuadricula2; // segundo objeto GridLayout
19
20    // constructor sin argumentos
21    public MarcoGridLayout()
22    {
23       super("Demostracion de GridLayout");
24       cuadricula1 = new GridLayout(2, 3, 5, 5); // 2 por 3; espacios de 5
25       cuadricula2 = new GridLayout(3, 2); // 3 por 2; sin espacios
26       contenedor = getContentPane();
27       setLayout(cuadricula1);
28       botones = new JButton[nombres.length];
29
30       for (int cuenta = 0; cuenta < nombres.length; cuenta++)
31       {
32          botones[cuenta] = new JButton(nombres[cuenta]);
33          botones[cuenta].addActionListener(this); // registra componente
                                                     de escucha
34          add(botones[cuenta]); // agrega boton a objeto JFrame
35       }
36    }
37
38    // maneja eventos de boton, alternando entre los esquemas
39    @Override
40    public void actionPerformed(ActionEvent evento)
41    {
```

Fig. 12.43 | GridLayout que contiene seis botones (parte 1 de 2).

```
42          if (alternar) // establece esquema con base en alternar
43             contenedor.setLayout(cuadricula2);
44          else
45             contenedor.setLayout(cuadricula1);
46
47          alternar = !alternar;
48          contenedor.validate(); // redistribuye el contenedor
49       }
50    } // fin de la clase MarcoGridLayout
```

Fig. 12.43 | `GridLayout` que contiene seis botones (parte 2 de 2).

```
1  // Fig. 12.44: DemoGridLayout.java
2  // Prueba de MarcoGridLayout.
3  import javax.swing.JFrame;
4
5  public class DemoGridLayout
6  {
7     public static void main(String[] args)
8     {
9        MarcoGridLayout marcoGridLayout = new MarcoGridLayout();
10       marcoGridLayout.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoGridLayout.setSize(300, 200);
12       marcoGridLayout.setVisible(true);
13    }
14 } // fin de la clase DemoGridLayout
```

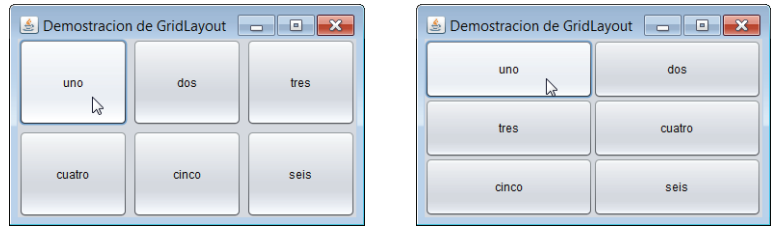


Fig. 12.44 | Prueba de `MarcoGridLayout`.

En las líneas 24 y 25 (figura 12.43) se crean dos objetos `GridLayout`. El constructor de `GridLayout` que se utiliza en la línea 24 especifica un objeto `GridLayout` con 2 filas, 3 columnas, 5 píxeles de espacio libre horizontal entre objetos `Component` en la cuadrícula y 5 píxeles de espacio libre vertical entre objetos `Component` en la cuadrícula. El constructor de `GridLayout` que se utiliza en la línea 25 especifica un objeto `GridLayout` con 3 filas y 2 columnas que utiliza el espacio libre predeterminado (1 píxel).

Los objetos `JButton` en este ejemplo se ordenan en un principio utilizando `cuadricula1` (que se establece para el panel de contenido en la línea 27, mediante el método `setLayout`). El primer componente se agrega a la primera columna de la primera fila. El siguiente componente se agrega a la segunda columna de la primera fila, y así en lo sucesivo. Cuando se oprime un objeto `JButton`, se hace una llamada al método `actionPerformed` (líneas 39 a 49). Todas las llamadas a `actionPerformed` alternan el esquema entre `cuadricula2` y `cuadricula1`, utilizando la variable `boolean` llamada `alternar` para determinar el siguiente esquema a establecer.

En la línea 48 se muestra otra manera para cambiar el formato a un contenedor para el cual haya cambiado el esquema. El método **`validate`** de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI que se muestran en pantalla.

12.19 Uso de paneles para administrar esquemas más complejos

Las GUI complejas (como la de la figura 12.1) requieren que cada componente se coloque en una ubicación exacta. A menudo constan de varios paneles, en donde los componentes de cada panel se ordenan en un esquema específico. La clase `JPanel` extiende a `JComponent`, y `JComponent` extiende a la clase `Container`, por lo que todo `JPanel` es un `Container`. Por lo tanto, todo objeto `JPanel` puede tener componentes, incluyendo otros paneles, los cuales se adjuntan mediante el método `add` de `Container`. La aplicación de las figuras 12.45 y 12.46 demuestra cómo puede usarse un objeto `JPanel` para crear un esquema más complejo, en el cual se coloquen varios objetos `JButton` en la región `SOUTH` de un esquema `BorderLayout`.

```
1  // Fig. 12.45: MarcoPanel.java
2  // Uso de un objeto JPanel para ayudar a distribuir los componentes.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8
9  public class MarcoPanel extends JFrame
10 {
11    private final JPanel panelBotones; // panel que contiene los botones
12    private final JButton[] botones;
13
14    // constructor sin argumentos
15    public MarcoPanel()
16    {
17       super("Demostracion de Panel");
18       botones = new JButton[5];
19       panelBotones = new JPanel();
20       panelBotones.setLayout(new GridLayout(1, botones.length));
21
22       // crea y agrega los botones
23       for (int cuenta = 0; cuenta < botones.length; cuenta++)
24       {
25          botones[cuenta] = new JButton("Boton " + (cuenta + 1));
26          panelBotones.add(botones[cuenta]); // agrega el botón al panel
27       }
28
29       add(panelBotones, BorderLayout.SOUTH); // agrega el panel a JFrame
30    }
31 } // fin de la clase MarcoPanel
```

Fig. 12.45 | `Jpanel` con cinco objetos `JButton`, en un esquema `GridLayout` adjunto a la región `SOUTH` de un esquema `BorderLayout`.

```
1  // Fig. 12.46: DemoPanel.java
2  // Prueba de MarcoPanel.
3  import javax.swing.JFrame;
4
5  public class DemoPanel extends JFrame
```

Fig. 12.46 | Prueba de `MarcoPanel` (parte 1 de 2).

```
6  {
7     public static void main(String[] args)
8     {
9        MarcoPanel marcoPanel = new MarcoPanel();
10       marcoPanel.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoPanel.setSize(450, 200);
12       marcoPanel.setVisible(true);
13    }
14 } // fin de la clase DemoPanel
```

Fig. 12.46 | Prueba de `MarcoPanel` (parte 2 de 2).

Una vez que el objeto `JPanel` llamado `panelBotones` se declara (línea 11 de la figura 12.45) y se crea (línea 19), en la línea 20 se establece el esquema de `panelBotones` a un `GridLayout` con una fila y cinco columnas (hay cinco objetos `JButton` en el arreglo `botones`). En las líneas 23 a 27 se agregan los cinco objetos `JButton` en el arreglo al objeto `JPanel`. En la línea 26 se agregan los botones directamente al objeto `JPanel` (la clase `JPanel` no tiene un panel de contenido, a diferencia de `JFrame`). En la línea 29 se utiliza el objeto `BorderLayout` predeterminado de `JFrame` para agregar `panelBotones` a la región `SOUTH`. Esta región tiene la misma altura que los botones en `panelBotones`. Un objeto `JPanel` ajusta su tamaño de acuerdo con los componentes que contiene. A medida que se agregan más componentes, el objeto `JPanel` *crece* (de acuerdo con las restricciones de su administrador de esquemas) para dar cabida a esos nuevos componentes. Ajuste el tamaño de la ventana para que vea cómo el administrador de esquemas afecta al tamaño de los objetos `JButton`.

12.20 JTextArea

Un objeto **`JTextArea`** proporciona un área para *manipular varias líneas de texto*. Al igual que la clase `JTextField`, `JTextArea` es una subclase de `JTextComponent`, el cual declara métodos comunes para objetos `JTextField`, `JTextArea` y varios otros componentes de GUI basados en texto.

La aplicación en las figuras 12.47 y 12.48 demuestra el uso de los objetos `JTextArea`. Un objeto `JTextArea` muestra texto que el usuario puede seleccionar. El otro no puede editarse, y se utiliza para mostrar el texto que seleccionó el usuario en el primer objeto `JTextArea`. A diferencia de los objetos `JTextField`, los objetos `JTextArea` no tienen eventos de acción, ya que al oprimir *Intro* mientras escribe en un objeto `JTextArea`, el cursor simplemente avanza a la siguiente línea. Al igual que con los objetos `JList` de selección múltiple (sección 12.13), un evento externo de otro componente de GUI indica cuándo se debe procesar el texto en un objeto `JTextArea`. Por ejemplo, al escribir un mensaje de correo electrónico, por lo general hacemos clic en un botón **Enviar** para enviar el texto del mensaje al destinatario. De manera similar, al editar un documento en un procesador de palabras, por lo general guardamos el archivo seleccionando un elemento de menú llamado **Guardar** o **Guardar como...**. En este programa, el botón **Copiar>>>** genera el evento externo que copia el texto seleccionado en el objeto `JTextArea` de la izquierda, y lo muestra en el objeto `JTextArea` de la derecha.

```
1  // Fig. 12.47: MarcoAreaTexto.java
2  // Copia el texto seleccionado de un área JText a otra.
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import javax.swing.Box;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10
11 public class MarcoAreaTexto extends JFrame
12 {
13    private final JTextArea areaTexto1; // muestra cadena de demostración
14    private final JTextArea areaTexto2; // el texto resaltado se copia aquí
15    private final JButton botonCopiar; // inicia el copiado de texto
16
17    // constructor sin argumentos
18    public MarcoAreaTexto()
19    {
20       super("Demostracion de JTextArea");
21       Box cuadro = Box.createHorizontalBox(); // crea un cuadro
22       String demo = "Esta es una cadena de\ndemostracion para\n" +
23          "ilustrar como copiar texto\nde un area de texto a \n" +
24          "otra, usando un\nevento externo\n";
25
26       areaTexto1 = new JTextArea(demo, 10, 15);
27       cuadro.add(new JScrollPane(areaTexto1)); // agrega panel de desplazamiento
28
29       botonCopiar = new JButton("Copiar >>>"); // crea botón para copiar
30       cuadro.add(botonCopiar); // agrega botón de copia al cuadro
31       botonCopiar.addActionListener(
32          new ActionListener() // clase interna anónima
33          {
34             // establece el texto en areaTexto2 con el texto seleccionado de
                areaTexto1
35             @Override
36             public void actionPerformed(ActionEvent evento)
37             {
38                areaTexto2.setText(areaTexto1.getSelectedText());
39             }
40          }
41       );
42
43       areaTexto2 = new JTextArea(10, 15);
44       areaTexto2.setEditable(false);
45       cuadro.add(new JScrollPane(areaTexto2)); // agrega panel de desplazamiento
46
47       add(cuadro); // agrega cuadro al marco
48    }
49 } // fin de la clase MarcoAreaTexto
```

Fig. 12.47 | Copiado de texto seleccionado, de un objeto `JTextArea` a otro.

```
1  // Fig. 12.48: DemoAreaTexto.java
2  // Prueba de MarcoAreaTexto.
3  import javax.swing.JFrame;
4
5  public class DemoAreaTexto
6  {
7     public static void main(String[] args)
8     {
9        MarcoAreaTexto marcoAreaTexto = new MarcoAreaTexto();
10       marcoAreaTexto.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       marcoAreaTexto.setSize(425, 200);
12       marcoAreaTexto.setVisible(true);
13    }
14 } // fin de la clase DemoAreaTexto
```

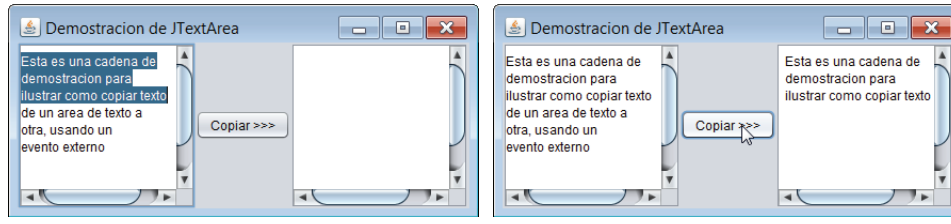


Fig. 12.48 | Prueba de `MarcoAreaTexto`.

En el constructor (líneas 18 a 48), la línea 21 crea un contenedor **`Box`** (paquete `javax.swing`) para organizar los componentes de la GUI. `Box` es una subclase de `Container` que utiliza un administrador de esquemas **`BoxLayout`** (que veremos con detalle en la sección 22.9) para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical. El método `static` **`createHorizontalBox`** de `Box` crea un objeto `Box` que ordena los componentes de izquierda a derecha, en el orden en el que se adjuntan.

En las líneas 26 y 43 se crean los objetos `JTextArea` llamados `areaTexto1` y `areaTexto2`. La línea 26 utiliza el constructor con tres argumentos de `JTextArea`, el cual recibe un objeto `String` que representa el texto inicial y dos valores `int` que especifican que el objeto `JTextArea` tiene 10 filas y 15 columnas. En la línea 43 se utiliza el constructor con dos argumentos de `JTextArea`, el cual especifica que el objeto `JTextArea` tiene 10 filas y 15 columnas. En la línea 26 se especifica que `demo` debe mostrarse como el contenido predeterminado del objeto `JTextArea`. Un objeto `JTextArea` no proporciona barras de desplazamiento si no puede mostrar su contenido completo. Por lo tanto, en la línea 27 se crea un objeto `JScrollPane`, se inicializa con `areaTexto1` y se adjunta al contenedor `cuadro`. En un objeto `JScrollPane` aparecen de manera predeterminada las barras de desplazamiento horizontal y vertical, según sea necesario.

En las líneas 29 a 41 se crea el objeto `JButton` llamado `botonCopiar` con la etiqueta "`Copiar >>>`", se agrega `botonCopiar` al contenedor `cuadro` y se registra el manejador de eventos para el evento `ActionEvent` de `botonCopiar`. Este botón proporciona el evento externo que determina cuándo debe copiar el programa el texto seleccionado en `areaTexto1` a `areaTexto2`. Cuando el usuario hace clic en `botonCopiar`, la línea 38 en `actionPerformed` indica que el método **`getSelectedText`** (que hereda `JTextArea` de `JTextComponent`) debe devolver el texto seleccionado de `areaTexto1`. Para seleccionar el texto, el usuario arrastra el ratón sobre el texto deseado para resaltarlo. El método `setText` cambia el texto en `areaTexto2` por la cadena que devuelve `getSelectedText`.

En las líneas 43 a 45 se crea `areaTexto2`, se establece su propiedad editable en `false` y se agrega al contenedor `box`. En la línea 47 se agrega `cuadro` al objeto `JFrame`. En la sección 12.18.2 vimos que el esquema predeterminado de un objeto `JFrame` es `BorderLayout`, y que el método `add` adjunta de manera predeterminada su argumento a la región `CENTER` de este esquema.

Cuando el texto llega al extremo derecho de un objeto `JTextArea`, puede recorrerse a la siguiente línea. A esto se le conoce como **ajuste de línea**. La clase `JTextArea` *no* ajusta las líneas de manera predeterminada.

Observación de apariencia visual 12.19

Para proporcionar la funcionalidad de ajuste de líneas para un objeto `JTextArea`, invoque el método `setLineWrap` de `JTextArea` con un argumento `true`.

Políticas de las barras de desplazamiento de `JScrollPane`

En este ejemplo se utiliza un objeto `JScrollPane` para proporcionar la capacidad de desplazamiento a un objeto `JTextArea`. De manera predeterminada, `JScrollPane` muestra las barras de desplazamiento *sólo* si se requieren. Puede establecer las **políticas de las barras de desplazamiento** horizontal y vertical de un objeto `JScrollPane` al momento de crearlo. Si un programa tiene una referencia a un objeto `JScrollPane`, puede usar los métodos `setHorizontalScrollBarPolicy` y `setVerticalScrollBarPolicy` de `JScrollPane` para modificar las políticas de las barras de desplazamiento en cualquier momento. La clase `JScrollPane` declara las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que *siempre debe aparecer una barra de desplazamiento*, las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que *debe aparecer una barra de desplazamiento sólo si es necesario* (los valores predeterminados), y las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que *nunca debe aparecer una barra de desplazamiento*. Si la política de la barra de desplazamiento horizontal se establece en `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, un objeto `JTextArea` adjunto al objeto `JScrollPane` ajustará las líneas de manera automática.

12.21 Conclusión

En este capítulo aprendió muchos componentes de la GUI, y cómo implementar el manejo de eventos. También aprendió las clases anidadas, las clases internas y las clases internas anónimas. Vio la relación especial entre un objeto de la clase interna y un objeto de su clase de nivel superior. Aprendió a utilizar diálogos `JOptionPane` para obtener datos de entrada de texto del usuario, y cómo mostrar mensajes a éste. También aprendió a crear aplicaciones que se ejecuten en sus propias ventanas. Hablamos sobre la clase `JFrame` y los componentes que permiten a un usuario interactuar con una aplicación. También le enseñamos cómo mostrar texto e imágenes al usuario. Vimos cómo personalizar los objetos `JPanel` para crear áreas de dibujo personalizadas, las cuales utilizará ampliamente en el siguiente capítulo. Vio cómo organizar los componentes en una ventana mediante el uso de los administradores de esquemas, y cómo crear GUI más complejas mediante el uso de objetos `JPanel` para organizar los componentes. Por último, aprendió acerca del componente `JTextArea`, en el cual un usuario puede introducir texto y una aplicación puede mostrarlo. En el capítulo 22 aprenderá acerca de los componentes de GUI más avanzados, como los botones deslizables, los menús y los administradores de esquemas más complicados. En el siguiente capítulo aprenderá a agregar gráficos a su aplicación de GUI. Los gráficos nos permiten dibujar figuras y texto con colores y estilos.

Resumen

Sección 12.1 Introducción

- Una interfaz gráfica de usuario (GUI; pág. 474) presenta un mecanismo amigable para que el usuario interactúe con una aplicación. Una GUI proporciona a una aplicación una "apariencia visual" única (pág. 474).
- Al dotar a las aplicaciones de componentes de interfaz de usuario que sean consistentes e intuitivos, los usuarios pueden familiarizarse con una nueva aplicación, de manera que pueden aprender a utilizarla con mayor rapidez.
- Las GUI se crean a partir de componentes de GUI (pág. 474); a éstos se les conoce algunas veces como controles o "widgets".

Sección 12.2 La apariencia visual Nimbus de Java

- A partir de la actualización 10 de Java SE 6, se incluye una nueva apariencia visual elegante multiplataforma, conocida como Nimbus (pág. 476).
- Para establecer Nimbus como predeterminada para todas las aplicaciones Java, cree un archivo de texto `swing.properties` en la carpeta `lib` de sus carpetas de instalación JDK y JRE. Coloque la siguiente línea de código en el archivo:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

- Para seleccionar Nimbus en cada aplicación por separado, coloque el siguiente argumento de línea de comandos después del comando `java` y antes del nombre de la aplicación, al momento de ejecutarla:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Sección 12.3 Entrada/salida simple basada en GUI con `JOptionPane`

- La mayoría de las aplicaciones utilizan ventanas o cuadros de diálogo (pág. 476) para interactuar con el usuario.
- La clase `JOptionPane` (pág. 476) del paquete `javax.swing` (pág. 474) proporciona cuadros de diálogo prefabricados para entrada y salida. El método `static showInputDialog` (477) de `JOptionPane` muestra un diálogo de entrada (pág. 476).
- Por lo general, un indicador utiliza las mayúsculas y minúsculas con el estilo de una oración: un estilo que usa mayúsculas sólo para la primera letra de la primera palabra del texto, a menos que alguna palabra sea un nombre propio.
- Un diálogo de entrada sólo puede introducir objetos `String`. Esto es común en la mayoría de los componentes de la GUI.
- El método `static showMessageDialog` (pág. 478) de `JOptionPane` muestra un diálogo de mensaje (pág. 476).

Sección 12.4 Generalidades de los componentes de Swing

- La mayoría de los componentes de GUI de Swing (pág. 474) se encuentran en el paquete `javax.swing`.
- En conjunto, a la apariencia y la forma en la que interactúa el usuario con la aplicación se les denomina la apariencia visual de esa aplicación. Los componentes de GUI de Swing nos permiten especificar una apariencia visual uniforme para una aplicación a través de todas las plataformas, o usar la apariencia visual personalizada de cada plataforma.
- Los componentes ligeros de Swing no están enlazados a los componentes actuales de GUI que soporta la plataforma subyacente en la que se ejecuta una aplicación.
- Varios componentes de Swing son componentes pesados (pág. 480), que requieren una interacción directa con el sistema de ventanas local (pág. 480), lo cual puede limitar su apariencia y funcionalidad.
- La clase `Component` (pág. 480) del paquete `java.awt` declara muchos de los atributos y comportamientos comunes para los componentes de GUI en los paquetes `java.awt` (pág. 479) y `javax.swing`.
- La clase `Container` (pág. 480) del paquete `java.awt` es una subclase de `Component`. Los objetos `Component` se adjuntan a los objetos `Container`, de manera que puedan organizarse y mostrarse en la pantalla.
- La clase `JComponent` (pág. 480) del paquete `javax.swing` es una subclase de `Container`. `JComponent` es la superclase de todos los componentes ligeros de Swing, y declara los atributos y comportamientos comunes.
- Algunas de las características comunes de `JComponent` son: una apariencia visual adaptable (pág. 480), teclas de método abreviado llamadas nemónicos (pág. 480), cuadros de información sobre herramientas (pág. 480), soporte para tecnologías de ayuda y soporte para la localización de la interfaz de usuario (pág. 480).

Sección 12.5 Presentación de texto e imágenes en una ventana

- La clase `JFrame` proporciona los atributos y comportamientos básicos de una ventana.
- Un objeto `JLabel` (pág. 481) muestra texto de sólo lectura, una imagen, o texto y una imagen. Por lo general, el texto en un objeto `JLabel` usa las mayúsculas con el estilo de una oración.
- Cada componente de una GUI debe adjuntarse a un contenedor, como una ventana creada con un objeto `JFrame` (pág. 483).
- Muchos IDE proporcionan herramientas de diseño de GUI (pág. 529), en las cuales podemos especificar el tamaño y la ubicación exactos de un componente mediante el uso del ratón; después el IDE genera el código de la GUI por nosotros.
- El método `setToolTipText` de `JComponent` (pág. 483) especifica la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre un componente ligero (pág. 480).
- El método `add` de `Container` adjunta un componente de GUI a un objeto `Container`.
- La clase `ImageIcon` (pág. 484) soporta varios formatos de imagen, incluyendo GIF, PNG y JPEG.
- El método `getClass` de la clase `Object` (pág. 484) obtiene una referencia al objeto `Class` que representa la declaración de la clase para el objeto en el que se hace la llamada al método.
- El método `getResource` de `Class` (pág. 484) devuelve la ubicación de su argumento en forma de URL. El método `getResource` usa el cargador de clases del objeto `Class` para determinar la ubicación del recurso.
- Las alineaciones horizontal y vertical de un objeto `JLabel` se pueden establecer mediante los métodos `setHorizontalAlignment` (pág. 484) y `setVerticalAlignment` (pág. 484), respectivamente.
- Los métodos `setText` (pág. 484) y `getText` (pág. 484) de `JLabel` establecen y obtienen el texto a mostrar en una etiqueta.
- Los métodos `setIcon` (pág. 484) y `getIcon` (pág. 484) de `JLabel` establecen y obtienen el objeto `Icon` (pág. 484) en una etiqueta.
- Los métodos `setHorizontalTextPosition` (pág. 484) y `setVerticalTextPosition` (pág. 484) de `JLabel` especifican la posición del texto en la etiqueta.
- El método `setDefaultCloseOperation` (pág. 485) de `JFrame`, con la constante `Frame.EXIT_ON_CLOSE` como argumento, indica que el programa debe terminar cuando el usuario cierre la ventana.
- El método `setSize` de `Component` (pág. 485) especifica la anchura y la altura de un componente.
- El método `setVisible` (pág. 485) de `Component` con el argumento `true` muestra un objeto `JFrame` en la pantalla.

Sección 12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas

- Las GUI se controlan por eventos, lo que significa que cuando el usuario interactúa con un componente de GUI, los eventos (pág. 485) controlan al programa para realizar las tareas.
- Un manejador de eventos (pág. 485) realiza una tarea en respuesta a un evento.
- La clase `JTextField` (pág. 485) extiende a la clase `JTextComponent` (pág. 485) del paquete `javax.swing.text`, que proporciona muchas características comunes para los componentes de Swing basados en texto. La clase `JPasswordField` (pág. 485) extiende a `JTextField` y agrega varios métodos específicos para el procesamiento de contraseñas.
- Un objeto `JPasswordField` (pág. 485) muestra que se están escribiendo caracteres a medida que el usuario los introduce, pero oculta los caracteres reales con caracteres de eco (pág. 485).
- Un componente recibe el enfoque (pág. 486) cuando el usuario hace clic sobre él.
- El método `setEditable` de `JTextComponent` (pág. 488) puede usarse para hacer que un campo de texto no pueda editarse.
- Para responder a un evento para un componente específico de la GUI, debemos crear una clase que represente al manejador de eventos e implementar una interfaz de escucha de eventos apropiada (pág. 488), después debemos registrar un objeto de la clase manejadora de eventos como el manejador de eventos (pág. 488).
- Las clases anidadas no `static` (pág. 488) se llaman clases internas, y se utilizan con frecuencia para el manejo de eventos.
- Un objeto de una clase no `static` interna (pág. 488) debe crearse mediante un objeto de la clase de nivel superior (pág. 488), que contenga a la clase interna.

- Un objeto de la clase interna puede acceder directamente a todas las variables y métodos de su clase de nivel superior.
- Una clase anidada que sea `static` no requiere un objeto de su clase de nivel superior, y no tiene una referencia implícita a un objeto de la clase de nivel superior.
- Cuando el usuario oprime *Intro* en un objeto `JTextField` o `JPasswordField`, el componente de la GUI genera un evento `ActionEvent` (pág. 489) que puede ser manejado por un objeto `ActionListener` (pág. 489) del paquete `java.awt.event`.
- El método `addActionListener` de `JTextField` (pág. 489) registra el manejador de eventos para un campo de texto de `ActionEvent`.
- El componente de GUI con el que interactúa el usuario es el origen del evento (pág. 490).
- Un objeto `ActionEvent` contiene información acerca del evento que acaba de ocurrir, como el origen del evento y el texto en el campo de texto.
- El método `getSource` de `ActionEvent` devuelve una referencia al origen del evento. El método `getActionCommand` (pág. 490) de `ActionEvent` devuelve el texto que escribió el usuario en un campo de texto o en la etiqueta de un objeto `JButton`.
- El método `getPassword` de `JPasswordField` (pág. 490) devuelve la contraseña que escribió el usuario.

Sección 12.7 Tipos de eventos comunes de la GUI e interfaces de escucha

- Para cada tipo de objeto evento, hay por lo general una interfaz de escucha de eventos que le corresponde y que especifica uno o más métodos manejadores de eventos, que deben declararse en la clase que implementa a la interfaz.

Sección 12.8 Cómo funciona el manejo de eventos

- Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus componentes de escucha registrados, llamando al método de manejo de eventos apropiado de cada componente de escucha.
- Todo componente de la GUI soporta varios tipos de eventos. Cuando ocurre un evento, éste se despacha (pág. 494) sólo a los componentes de escucha de eventos del tipo apropiado.

Sección 12.9 `JButton`

- Un botón es un componente en el que el usuario hace clic para desencadenar cierta acción. Todos los tipos de botones son subclases de `AbstractButton` (pág. 495; paquete `javax.swing`). Por lo general, las etiquetas de los botones (pág. 495) usan mayúsculas para la primera letra de cada palabra representativa, como en un libro en inglés (pág. 495).
- Los botones de comandos (pág. 495) se crean con la clase `JButton`.
- Un objeto `JButton` puede mostrar un objeto `Icon`. Un objeto `JButton` también puede tener un icono de sustitución (pág. 497), que es un objeto `Icon` que se muestra cuando el usuario coloca el ratón sobre el botón.
- El método `setRolloverIcon` (pág. 498) de la clase `AbstractButton` especifica la imagen a mostrar en un botón, cuando el usuario coloca el ratón sobre él.

Sección 12.10 Botones que mantienen el estado

- Hay tres tipos de botones de estado de Swing: `JToggleButton` (pág. 498), `JCheckBox` (pág. 498) y `JRadioButton` (pág. 498).
- Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton`.
- El método `setFont` (de la clase `Component`) (pág. 500) establece el tipo de letra de un componente a un nuevo objeto de la clase `Font` (pág. 500) del paquete `java.awt`.
- Al hacer clic en un objeto `JCheckBox`, ocurre un evento `ItemEvent` (pág. 501), que puede manejarse mediante un objeto `ItemListener` (pág. 501), que define al método `itemStateChanged` (pág. 501). El método `addItemListener` registra el componente de escucha para un objeto `JCheckBox` o `JRadioButton`.
- El método `isSelected` de `JCheckBox` determina si un objeto `JCheckBox` está seleccionado.

- Los objetos `JRadioButton` tienen dos estados: seleccionado y no seleccionado. Por lo general, los botones de opción (pág. 495) aparecen como un grupo (pág. 501), en el cual sólo puede seleccionarse un botón a la vez.
- Los objetos `JRadioButton` se utilizan para representar opciones mutuamente excluyentes (pág. 501).
- La relación lógica entre los objetos `JRadioButton` se mantiene mediante un objeto `ButtonGroup` (pág. 501).
- El método `add` de `ButtonGroup` (pág. 504) asocia a cada objeto `JRadioButton` con un objeto `ButtonGroup`. Si se agrega más de un objeto `JRadioButton` seleccionado a un grupo, el primer objeto `JRadioButton` seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.
- Los objetos `JRadioButton` generan eventos `ItemEvent` cuando se hace clic sobre ellos.

Sección 12.11 `JComboBox`: uso de una clase interna anónima para el manejo de eventos

- Un objeto `JComboBox` (pág. 504) proporciona una lista de elementos, de los cuales el usuario puede seleccionar uno. Los objetos `JComboBox` generan eventos `ItemEvent`.
- Cada elemento en un objeto `JComboBox` tiene un índice (pág. 507). El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado cuando se muestra el objeto `JComboBox`.
- El método `setMaximumRowCount` de `JComboBox` (pág. 507) establece el máximo número de elementos a mostrar cuando el usuario haga clic en el objeto `JComboBox`.
- Una clase interna anónima (pág. 507) es una clase interna sin nombre y por lo general aparece dentro de la declaración de un método. Un objeto de la clase interna anónima debe crearse en el punto en el que se declara la clase.
- El método `getSelectedIndex` de `JComboBox` (pág. 508) devuelve el índice del elemento seleccionado.

Sección 12.12 `JList`

- Un objeto `JList` muestra una serie de elementos, de los cuales el usuario puede seleccionar uno o más. La clase `JList` soporta las listas de selección simple (pág. 508) y de selección múltiple.
- Cuando el usuario hace clic en un elemento de un objeto `JList`, ocurre un evento `ListSelectionEvent` (pág. 508). El método `addListSelectionListener` de `JList` (pág. 510) registra un objeto `ListSelectionListener` (pág. 510) para los eventos de selección de un objeto `JList`. Un objeto `ListSelectionListener` del paquete `javax.swing.event` (pág. 492) debe implementar el método `valueChanged`.
- El método `setVisibleRowCount` de `JList` (pág. 510) especifica el número de elementos visibles en la lista.
- El método `setSelectionMode` de `JList` (pág. 510) especifica el modo de selección de una lista.
- Un objeto `JList` se puede adjuntar a un `JScrollPane` (pág. 510) para proveer una barra de desplazamiento para ese objeto `JList`.
- El método `getContentPane` de `JFrame` (pág. 510) devuelve una referencia al panel de contenido de `JFrame`, en donde se muestran los componentes de la GUI.
- El método `getSelectedIndex` de `JList` (pág. 511) devuelve el índice del elemento seleccionado.

Sección 12.13 Listas de selección múltiple

- Una lista de selección múltiple (pág. 511) permite al usuario seleccionar muchos elementos de un objeto `JList`.
- El método `setFixedCellWidth` de `JList` (pág. 513) establece la anchura de un objeto `JList`. El método `setFixedCellHeight` (pág. 513) establece la altura de cada elemento en un objeto `JList`.
- Por lo general, un evento externo (pág. 513) generado por otro componente de la GUI (como un `JButton`) especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`.
- El método `setListData` de `JList` (pág. 513) establece los elementos a mostrar en un objeto `JList`. El método `getSelectedValues` de `JList` (pág. 513) devuelve un arreglo de objetos `Object` que representan los elementos seleccionados en un objeto `JList`.

Sección 12.14 Manejo de eventos de ratón

- Las interfaces de escucha de eventos `MouseListener` (pág. 513) y `MouseMotionListener` (pág. 513) se utilizan para manejar los eventos del ratón (pág. 513). Estos eventos se pueden atrapar para cualquier componente de la GUI que extienda a `Component`.

- La interfaz `MouseInputListener` (pág. 513) del paquete `javax.swing.event` extiende a las interfaces `MouseListener` y `MouseMotionListener` para crear una sola interfaz que contenga a todos sus métodos.
- Cada uno de los métodos manejadores de eventos del ratón recibe un objeto `MouseEvent` (pág. 494), el cual contiene información acerca del evento, incluyendo las coordenadas *x* y *y* de la ubicación en donde ocurrió el evento. Estas coordenadas se miden empezando desde la esquina superior izquierda del componente de la GUI en donde ocurrió el evento.
- Los métodos y constantes de la clase `InputEvent` (pág. 514; superclase de `MouseEvent`) permiten a una aplicación determinar cuál botón oprimió el usuario.
- La interfaz `MouseWheelListener` (pág. 515) permite a las aplicaciones responder a los eventos de la rueda de un ratón.

Sección 12.15 Clases adaptadoras

- Una clase adaptadora (pág. 518) implementa a una interfaz y proporciona implementaciones predeterminadas de sus métodos. Al extender una clase adaptadora, podemos sobrescribir sólo los métodos que necesitamos.
- El método `getClickCount` de `MouseEvent` (pág. 521) devuelve el número de clics consecutivos de los botones del ratón. Los métodos `isMetaDown` (pág. 528) e `isAltDown` (pág. 521) determinan cuál botón del ratón oprimió el usuario.

Sección 12.16 Subclase de `JPanel` para dibujar con el ratón

- El método `paintComponent` de `JComponent` (pág. 522) se llama cuando se muestra un componente ligero de Swing. Al sobrescribir este método, puede especificar cómo dibujar figuras usando las herramientas de gráficos de Java.
- Al sobrescribir el método `paintComponent`, hay que llamar a la versión de la superclase como la primera instrucción en el cuerpo.
- Las subclases de `JComponent` soportan la transparencia. Cuando un componente es opaco (pág. 522), `paintComponent` borra el fondo del componente antes de mostrarlo en pantalla.
- La transparencia de un componente ligero de Swing puede establecerse con el método `setOpaque` (pág. 522; un argumento `false` indica que el componente es transparente).
- La clase `Point` (pág. 523) del paquete `java.awt` representa una coordenada *x*-*y*.
- La clase `Graphics` (pág. 523) se utiliza para dibujar.
- El método `getPoint` de `MouseEvent` (pág. 524) obtiene el objeto `Point` en donde ocurrió un evento de ratón.
- El método `repaint` (pág. 524), heredado directamente de la clase `Component`, indica que un componente debe actualizarse en la pantalla lo más pronto posible.
- El método `paintComponent` recibe un parámetro `Graphics`, y se llama automáticamente cada vez que un componente ligero necesita mostrarse en la pantalla.
- El método `fillOval` de `Graphics` (pág. 524) dibuja un óvalo relleno. Los primeros dos argumentos son la coordenada *x* superior izquierda y la coordenada *y* superior izquierda del área rectangular delimitadora, mientras que las últimas dos coordenadas representan la anchura y la altura del área rectangular.

Sección 12.17 Manejo de eventos de teclas

- La interfaz `KeyListener` (pág. 494) se utiliza para manejar eventos de teclas (pág. 494) que se generan cuando se oprimen y sueltan las teclas en el teclado. El método `addKeyListener` de la clase `Component` (pág. 525) registra un objeto `KeyListener`.
- El método `getKeyCode` (pág. 528) de `KeyEvent` (pág. 494) obtiene el código de tecla virtual (pág. 528) de la tecla oprimida. La clase `KeyEvent` mantiene un conjunto de constantes de código de tecla virtual que representa a todas las teclas en el teclado.
- El método `getKeyText` (pág. 528) de `KeyEvent` devuelve una cadena que contiene el nombre de la tecla que se oprimió.
- El método `getKeyChar` (pág. 528) de `KeyEvent` obtiene el valor Unicode del carácter escrito.
- El método `isActionKey` (pág. 528) de `KeyEvent` determina si la tecla en un evento fue una tecla de acción (pág. 525).
- El método `getModifiers` (pág. 528) de `InputEvent` determina si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla.

- El método `getKeyModifiersText` (pág. 528) de `KeyEvent` produce una cadena que contiene los nombres de las teclas modificadoras que se oprimieron.

Sección 12.18 Introducción a los administradores de esquemas

- Los administradores de esquemas (pág. 528) ordenan los componentes de la GUI en un contenedor, para fines de presentación.
- Todos los administradores de esquemas implementan la interfaz `LayoutManager` (pág. 528) del paquete `java.awt`.
- El método `setLayout` de la clase `Container` especifica el esquema de un contenedor.
- `FlowLayout` coloca componentes de izquierda a derecha, en el orden en el que se agregaron al contenedor. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. `FlowLayout` permite a los componentes de la GUI alinearse a la izquierda, al centro (el valor predeterminado) y a la derecha.
- El método `setAlignment` (pág. 532) de `FlowLayout` cambia la alineación para un objeto `FlowLayout`.
- `BorderLayout` (pág. 532; el predeterminado para un objeto `JFrame`) ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. `NORTH` corresponde a la parte superior del contenedor.
- Un `BorderLayout` limita a un objeto `Container` para que contenga cuando mucho cinco componentes; uno en cada región.
- `GridLayout` (pág. 536) divide un contenedor en una cuadrícula de filas y columnas.
- El método `validate` (pág. 537) de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI que se muestran en pantalla.

Sección 12.19 Uso de paneles para administrar esquemas más complejos

- Las GUI complejas constan a menudo de varios paneles con distintos esquemas. Cada `JPanel` puede tener componentes, incluyendo otros paneles, los cuales se adjuntan mediante el método `add` de `Container`.

Sección 12.20 `JTextArea`

- Un objeto `JTextArea` (pág. 539; una subclase de `JTextComponent`) puede contener varias líneas de texto.
- La clase `Box` (pág. 540) es una subclase de `Container` que utiliza un administrador de esquemas `BoxLayout` (pág. 541) para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical.
- El método `static createHorizontalBox` (pág. 541) de `Box` crea un objeto `Box` que ordena los componentes de izquierda a derecha, en el orden en el que se adjuntan.
- El método `getSelectedText` (pág. 541) devuelve el texto seleccionado de un objeto `JTextArea`.
- Podemos establecer las políticas de las barras de desplazamiento horizontal y vertical (pág. 542) de un objeto `JScrollPane` al momento de crearlo. Los métodos `setHorizontalScrollBarPolicy` (pág. 542) y `setVerticalScrollBarPolicy` (pág. 542) de `JScrollPane` pueden usarse para modificar las políticas de las barras de desplazamiento en cualquier momento.

Ejercicios de autoevaluación

12.1 Complete las siguientes oraciones:

a) El método _____ es llamado cuando el ratón se mueve sin oprimir los botones y un componente de escucha de eventos está registrado para manejar el evento.
b) El texto que no puede ser modificado por el usuario se llama texto _____.
c) Un _____ ordena los componentes de la GUI en un objeto `Container`.
d) El método `add` para adjuntar componentes de la GUI es un método de la clase _____.
e) GUI es un acrónimo para _____.
f) El método _____ se utiliza para especificar el administrador de esquemas para un contenedor.
g) Una llamada al método `mouseDragged` va precedida por una llamada al método _____ y va seguida de una llamada al método _____.
h) La clase _____ contiene métodos que muestran diálogos de mensaje y diálogos de entrada.
i) Un diálogo de entrada capaz de recibir entrada del usuario se muestra con el método _____ de la clase _____.

j) Un diálogo capaz de mostrar un mensaje al usuario se muestra con el método _____ de la clase _____.

k) `JTextField` y `JTextArea` extienden directamente a la clase _____.

12.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) `BorderLayout` es el administrador de esquemas predeterminado para un panel de contenido de `JFrame`.
b) Cuando el cursor del ratón se mueve hacia los límites de un componente de la GUI, se hace una llamada al método `mouseOver`.
c) Un objeto `JPanel` no puede agregarse a otro `JPanel`.
d) En un esquema `BorderLayout`, dos botones que se agreguen a la región `NORTH` se mostrarán uno al lado del otro.
e) Se puede agregar un máximo de cinco componentes a un `BorderLayout`.
f) Las clases internas no pueden acceder a los miembros de la clase que las encierra.
g) El texto de un objeto `JTextArea` siempre es de sólo lectura.
h) La clase `JTextArea` es una subclase directa de la clase `Component`.

12.3 Encuentre los errores en cada una de las siguientes instrucciones y explique cómo corregirlos.

a)
```
nombreBoton = JButton("Leyenda");
```
b)
```
JLabel unaEtiqueta, JLabel;
```
c)
```
campoTexto = new JTextField(50, "Texto predeterminado");
```
d)
```
setLayout(new BorderLayout());
boton1 = new JButton("Estrella del norte");
boton2 = new JButton("Polo sur");
add(boton1);
add(boton2);
```

Respuestas a los ejercicios de autoevaluación

12.1 a) `mouseMoved`. b) no editable (de sólo lectura). c) administrador de esquemas. d) `Container`. e) interfaz gráfica de usuario. f) `setLayout`. g) `mousePressed`, `mouseReleased`. h) `JOptionPane`. i) `showInputDialog`, `JOptionPane`. j) `showMessageDialog`, `JOptionPane`. k) `JTextComponent`.

12.2 a) Verdadero.
b) Falso. Se hace una llamada al método `mouseEntered`.
c) Falso. Un `JPanel` puede agregarse a otro `JPanel`, ya que `JPanel` es una subclase indirecta de `Component`. Por lo tanto, un `JPanel` es un `Component`. Cualquier `Component` puede agregarse a un `Container`.
d) Falso. Sólo se mostrará el último botón que se agregue. Recuerde que sólo debe agregarse un componente a cada región en un esquema `BorderLayout`.
e) Verdadero. [*Nota:* se pueden agregar paneles que contienen varios componentes en cada región].
f) Falso. Las clases internas tienen acceso a todos los miembros de la declaración de la clase que las encierra.
g) Falso. Los objetos `JTextArea` pueden editarse de manera predeterminada.
h) Falso. `JTextArea` se deriva de la clase `JTextComponent`.

12.3 a) se necesita `new` para crear un objeto.
b) `JLabel` es el nombre de una clase y no puede utilizarse como nombre de variable.
c) Los argumentos que se pasan al constructor están invertidos. El objeto `String` debe pasarse primero.
d) Se ha establecido `BorderLayout` y los componentes se agregarán sin especificar la región, por lo que ambos se agregarán a la región central. Las instrucciones `add` apropiadas serían:
```
add(boton1, BorderLayout.NORTH);
add(boton2, BorderLayout.SOUTH);
```

Ejercicios

12.4 Complete las siguientes oraciones:

a) La clase `JTextField` extiende directamente a la clase _____.

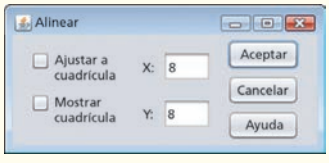
b) El método _____ de `Container` adjunta un componente de la GUI a un contenedor.
c) El método _____ es llamado cuando se suelta uno de los botones del ratón (sin mover el ratón).
d) La clase _____ se utiliza para crear un grupo de objetos `JRadioButton`.

12.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Sólo puede usarse un administrador de esquemas por cada objeto `Container`.
b) En un esquema `BorderLayout`, los componentes de la GUI pueden agregarse a un objeto `Container` en cualquier orden.
c) Los objetos `JRadioButton` proporcionan una serie de opciones mutuamente excluyentes (es decir, sólo uno puede ser `true` en un momento dado).
d) El método `setFont` de `Graphics` se utiliza para establecer el tipo de letra para los campos de texto.
e) Un objeto `JList` muestra una barra de desplazamiento si hay más elementos en la lista de los que puedan mostrarse en pantalla.
f) Un objeto `Mouse` tiene un método llamado `mouseDragged`.

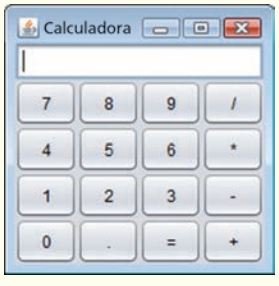
12.6 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Un objeto `JPanel` es un objeto `JComponent`.
b) Un objeto `JPanel` es un objeto `Component`.
c) Un objeto `JLabel` es un objeto `Container`.
d) Un objeto `JList` es un objeto `JPanel`.
e) Un objeto `AbstractButton` es un objeto `JButton`.
f) Un objeto `JTextField` es un objeto `Object`.
g) `ButtonGroup` es una subclase de `JComponent`.

12.7 Encuentre los errores en cada una de las siguientes líneas de código y explique cómo corregirlos.
a) `import javax.swing.JFrame`
b) `objetoPanel.GridLayout(8, 8);`
c) `contenedor.setLayout(new FlowLayout(FlowLayout.DEFAULT));`
d) `contenedor.add(botonEste, EAST);`

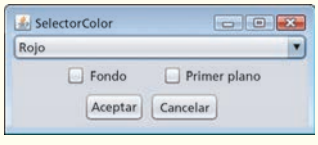
12.8 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



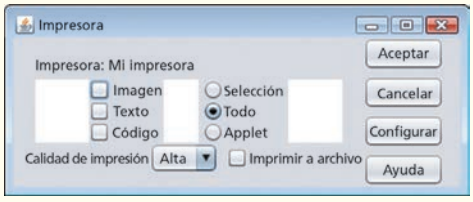
12.9 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



12.10 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



12.11 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



12.12 ***(Conversión de temperatura)*** Escriba una aplicación de conversión de temperatura, que convierta de grados Fahrenheit a Centígrados. La temperatura en grados Fahrenheit deberá introducirse desde el teclado (mediante un objeto `JTextField`). Debe usarse un objeto `JLabel` para mostrar la temperatura convertida. Use la siguiente fórmula para la conversión:

$$Centígrados = \frac{5}{9} \times (Fahrenheit - 32)$$

12.13 ***(Modificación a la conversión de temperatura)*** Mejore la aplicación de conversión de temperatura del ejercicio 12.12, agregando la escala de temperatura Kelvin. Además, la aplicación debe permitir al usuario realizar conversiones entre dos escalas cualesquiera. Use la siguiente fórmula para la conversión entre Kelvin y Centígrados (además de la fórmula del ejercicio 12.12):

Kelvin = *Centígrados* + `273.15`

12.14 ***(Juego: adivine el número)*** Escriba una aplicación que juegue a "adivinar el número" de la siguiente manera: su aplicación debe elegir el número a adivinar, seleccionando un entero al azar en el rango de 1 a 1000. La aplicación entonces deberá mostrar lo siguiente en una etiqueta:

```
Tengo un numero entre 1 y 1000. Puede usted adivinarlo?
Por favor escriba su primer intento.
```

Debe usarse un objeto `JTextField` para introducir el intento. A medida que se introduzca cada intento, el color de fondo deberá cambiar ya sea a rojo o azul. Rojo indica que el usuario se está "acercando" y azul indica que el usuario se está "alejando". Un objeto `JLabel` deberá mostrar el mensaje "`Demasiado alto`" o "`Demasiado bajo`" para ayudar al usuario a tratar de adivinar correctamente el número. Cuando el usuario adivine el número, deberá mostrarse el mensaje "`Correcto!`", y el objeto `JTextField` utilizado para la entrada deberá cambiar para que no pueda editarse. Debe proporcionarse un objeto `JButton` para permitir al usuario jugar de nuevo. Cuando se haga clic en el objeto `JButton`, deberá generarse un nuevo número aleatorio y el objeto `JTextField` de entrada deberá cambiar para poder editarse otra vez.

12.15 ***(Mostrar eventos)*** A menudo es conveniente mostrar los eventos que ocurren durante la ejecución de una aplicación. Esto puede ayudarle a comprender cuándo ocurren los eventos y cómo se generan. Escriba una aplicación que permita al usuario generar y procesar cada uno de los eventos descritos en este capítulo. La aplicación deberá proporcionar métodos de las interfaces `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` y `KeyListener`, para mostrar mensajes cuando ocurran los eventos. Use el método `toString` para convertir los objetos evento que se reciban en cada manejador de eventos, en un objeto `String` que pueda mostrarse en pantalla. El método `toString` crea un objeto `String` que contiene toda la información del objeto evento.

12.16 ***(Juego de Craps basado en GUI)*** Modifique la aplicación de la sección 6.10 para proporcionar una GUI que permita al usuario hacer clic en un objeto `JButton` para tirar los dados. La aplicación debe también mostrar cuatro objetos `JLabel` y cuatro objetos `JTextField`, con un objeto `JLabel` para cada objeto `JTextField`. Los objetos `JTextField` deben usarse para mostrar los valores de cada dado, y la suma de los dados después de cada tiro. El punto debe mostrarse en el cuarto objeto `JTextField` cuando el usuario no gane o pierda en el primer tiro, y debe seguir mostrándose hasta que el usuario pierda el juego.

(Opcional) Ejercicio del ejemplo práctico de GUI y gráficos: expansión de la interfaz

12.17 ***(Aplicación de dibujo interactiva)*** En este ejercicio, implementará una aplicación de GUI que utiliza la jerarquía `MiFigura` del ejercicio 10.2 del ejemplo práctico de GUI y gráficos, para crear una aplicación de dibujo interactiva. Debe crear dos clases para la GUI y proporcionar una clase de prueba para iniciar la aplicación. Las clases de la jerarquía `MiFigura` no requieren modificaciones adicionales.

La primera clase a crear es una subclase de `JPanel` llamada `PanelDibujo`, la cual representa el área en la cual el usuario dibuja las figuras. La clase `PanelDibujo` debe tener las siguientes variables de instancia:

a) Un arreglo llamado `figuras` de tipo `MiFigura`, que almacene todas las figuras que dibuje el usuario.
b) Una variable entera llamada `cuentaFiguras`, que cuente el número de figuras en el arreglo.
c) Una variable entera llamada `tipoFigura`, que determine el tipo de la figura a dibujar.
d) Un objeto `MiFigura` llamado `figuraActual`, que represente la figura actual que está dibujando el usuario.
e) Un objeto `Color` llamado `colorActual`, que represente el color del dibujo actual.
f) Una variable `boolean` llamada `figuraRellena`, que determine si se va a dibujar una figura rellena.
g) Un objeto `JLabel` llamado `etiquetaEstado`, que represente a la barra de estado. Esta barra deberá mostrar las coordenadas de la posición actual del ratón.

La clase `PanelDibujo` también debe declarar los siguientes métodos:

a) El método sobrescrito `paintComponent`, que dibuja las figuras en el arreglo. Use la variable de instancia `cuentaFiguras` para determinar cuántas figuras hay que dibujar. El método `paintComponent` también debe llamar al método `draw` de `figuraActual`, siempre y cuando `figuraActual` no sea `null`.
b) Métodos *establecer* para `tipoFigura`, `colorActual` y `figuraRellena`.
c) El método `borrarUltimaFigura` debe borrar la última figura dibujada, disminuyendo la variable de instancia `cuentaFiguras`. Asegúrese de que `cuentaFiguras` nunca sea menor que cero.
d) El método `borrarDibujo` debe eliminar todas las figuras en el dibujo actual, estableciendo `cuentaFiguras` en cero.

Los métodos `borrarUltimaFigura` y `borrarDibujo` deben llamar al método `repaint` (heredado de `Jpanel`) para actualizar el dibujo en el objeto `PanelDibujo`, indicando que el sistema nunca debe llamar al método `paintComponent`.

La clase `PanelDibujo` también debe proporcionar el manejo de eventos, para permitir al usuario dibujar con el ratón. Cree una clase interna individual que extienda a `MouseAdapter` e implemente a `MouseMotionListener` para manejar todos los eventos de ratón en una clase.

En la clase interna, sobrescriba el método `mousePressed` de manera que asigne a `figuraActual` una nueva figura del tipo especificado por `tipoFigura`, y que inicialice ambos puntos con la posición del ratón. A continuación, sobrescriba el método `mouseReleased` para terminar de dibujar la figura actual y colocarla en el arreglo. Establezca el segundo punto de `figuraActual` con la posición actual del ratón y agregue `figuraActual` al arreglo. La variable de instancia `cuentaFiguras` determina el índice de inserción. Establezca `figuraActual` a `null` y llame al método `repaint` para actualizar el dibujo con la nueva figura.

Sobrescriba el método `mouseMoved` para establecer el texto de `etiquetaEstado`, de manera que muestre las coordenadas del ratón; esto actualizará la etiqueta con las coordenadas cada vez que el usuario mueva (pero no arrastre) el ratón dentro del objeto `PanelDibujo`. A continuación, sobrescriba el método `mouseDragged` de manera que establezca el segundo punto de `figuraActual` con la posición actual del ratón y llame al método `repaint`. Esto permitirá al usuario ver la figura mientras arrastra el ratón. Además, actualice el objeto `JLabel` en `mouseDragged` con la posición actual del ratón.

Cree un constructor para `PanelDibujo` que tenga un solo parámetro `JLabel`. En el constructor, inicialice `etiquetaEstado` con el valor que se pasa al parámetro. Además, inicialice el arreglo `figuras` con 100 entradas, `cuentaFiguras` con 0, `tipoFigura` con el valor que represente a una línea, `figuraActual` con `null` y `colorActual` con `Color.BLACK`. El constructor deberá entonces establecer el color de fondo del objeto `PanelDibujo` a `Color.WHITE` y registrar a `MouseListener` y `MouseMotionListener`, de manera que el objeto `JPanel` maneje los eventos de ratón en forma apropiada.

A continuación, cree una subclase de `JFrame` llamada `MarcoDibujo`, que proporcione una GUI que permita al usuario controlar varios aspectos del dibujo. Para el esquema del objeto `MarcoDibujo`, recomendamos `BorderLayout`,

con los componentes en la región `NORTH`, el panel de dibujo principal en la región `CENTER` y una barra de estado en la región `SOUTH`, como en la figura 12.49. En el panel superior, cree los componentes que se listan a continuación. El manejador de eventos de cada componente deberá llamar al método apropiado en la clase `PanelDibujo`.

a) Un botón para deshacer la última figura que se haya dibujado.
b) Un botón para borrar todas las figuras del dibujo.
c) Un cuadro combinado para seleccionar el color de los 13 colores predefinidos.
d) Un cuadro combinado para seleccionar la figura a dibujar.
e) Una casilla de verificación que especifique si una figura debe estar rellena o sin relleno.

Declare y cree los componentes de la interfaz en el constructor de `MarcoDibujo`. Necesitará crear la barra de estado `JLabel` antes de crear el objeto `PanelDibujo`, de manera que pueda pasar el objeto `JLabel` como argumento para el constructor de `PanelDibujo`. Por último, cree una clase de prueba para inicializar y mostrar el objeto `MarcoDibujo` para ejecutar la aplicación.

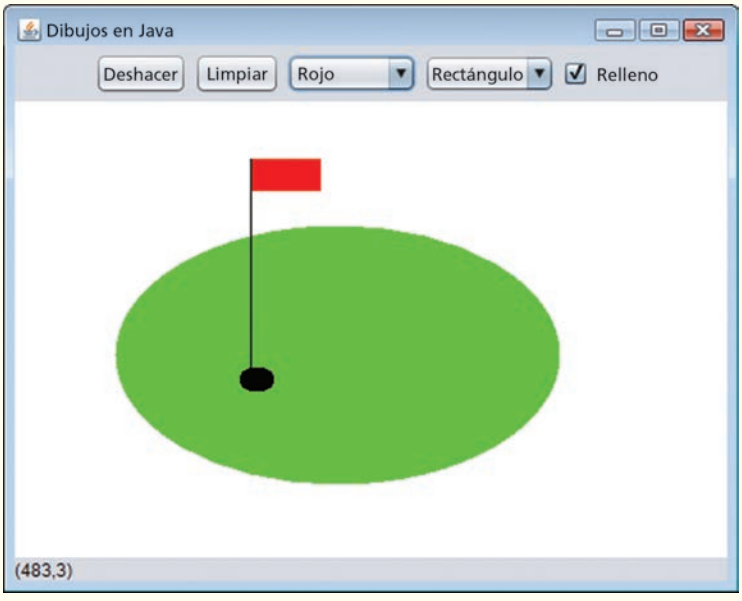


Fig. 12.49 | Interfaz para dibujar figuras.

12.18 ***(Versión del ejemplo práctico del ATM basada en GUI)*** Vuelva a implementar el ejemplo práctico del ATM de los capítulos 33 y 34 como una aplicación basada en GUI. Use componentes de GUI para crear un diseño aproximado de la interfaz que se muestra en la figura 33.1. Para el dispensador de efectivo y la ranura de depósito, use objetos `JButton` etiquetados como **Recoger efectivo** e **Insertar sobre**. Esto permitirá a la aplicación recibir eventos que indiquen cuando el usuario toma el efectivo e inserta un sobre de depósito, respectivamente.

Marcando la diferencia

12.19 ***(Ecofont)*** Ecofont (`www.ecofont.eu/ecofont_en.html`) –desarrollada por SPRANQ (una compañía con sede en los Países Bajos)— es un tipo de letra de computadora gratuito de código fuente abierto, diseñado para reducir hasta en un 20% la cantidad de tinta utilizada para imprimir, con lo cual se reduce también el número de cartuchos de tinta utilizados y el impacto ambiental de los procesos de manufactura y envío (se usa menos energía, menos combustible para el envío, etcétera). El tipo de letra, basado en Verdana sans-serif, tiene pequeños "orificios" circulares en las letras que no son visibles en tamaños más pequeños, como el tipo de 9 o 10 puntos de uso frecuente. Descargue

Ecofont, después instale el archivo de tipo de letra `Spranq_eco_sans_regular.ttf` usando las instrucciones del sitio Web de Ecofont. A continuación, desarrolle un programa basado en GUI que le permita escribir una cadena de texto para visualizarse en Ecofont. Cree botones **Aumentar tamaño de letra** y **Reducir tamaño de letra** que le permitan escalar hacia arriba o hacia abajo, un punto a la vez. Empiece con un tamaño de tipo de letra predeterminado de 9 puntos. A medida que vaya escalando hacia arriba, podrá ver con más claridad los orificios en las letras. A medida que vaya escalando hacia abajo, los orificios serán menos aparentes. ¿Cuál es el menor tamaño de tipo de letra en el que empezó a notar los orificios?

12.20 ***(Tutor de mecanografía: optimización de una habilidad crucial en la era de las computadoras)*** Escribir con rapidez y en forma correcta es una habilidad esencial para trabajar de manera efectiva con las computadoras e Internet. En este ejercicio, creará una aplicación de GUI que pueda ayudar a los usuarios a aprender a "escribir al tacto" (es decir, escribir correctamente sin ver el teclado). La aplicación deberá mostrar un *teclado virtual* (figura 12.50) y deberá permitir al usuario ver lo que escribe en la pantalla, sin tener que ver el *teclado real*. Use objetos `JButton` para representar las teclas. A medida que el usuario oprima cada tecla, la aplicación deberá resaltar el objeto `JButton` correspondiente en la GUI, y agregará el carácter a un objeto `JTextArea` que mostrará lo que ha escrito el usuario en un momento dado. [*Sugerencia:* para resaltar un `JButton`, use su método `setBackground` para cambiar su color de fondo. Cuando se libere la tecla, restablezca su color de fondo original. Puede obtener el color de fondo original del objeto `JButton` mediante el método `getBackground` antes de cambiar su color].

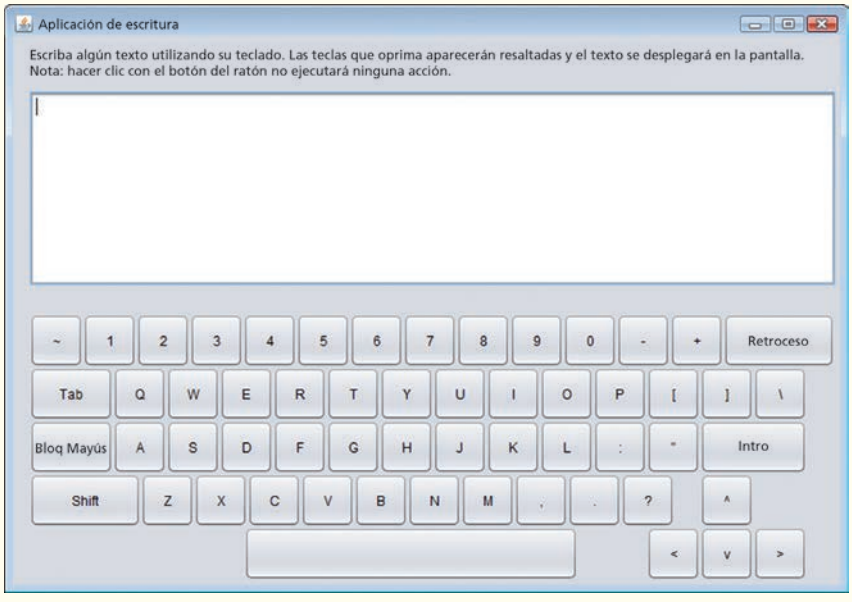


Fig. 12.50 | Tutor de mecanografía.

Para probar su programa, escriba un pangrama: una frase que contiene todas las letras del alfabeto por lo menos una vez, como "El veloz murciélago hindú comía feliz cardillo y kiwi. La cigüeña tocaba el saxofón detrás del palenque de paja". Encontrará más pangramas en Web.

Para que el programa sea más interesante, podría monitorear la precisión del usuario. Podría hacer que el usuario escribiera frases específicas que tenga pregrabadas en su programa, y que muestre en la pantalla por encima del teclado virtual. Podría llevar la cuenta de cuántas pulsaciones de teclas ha escrito correctamente el usuario, y cuántas escribió en forma incorrecta. Podría también llevar la cuenta de cuáles son las teclas que se le dificultan al usuario, y mostrar un informe en donde aparezcan esas teclas.