CHAPTER

14

# USING LAYOUT MANAGERS AND THE EVENT MODEL

**In this chapter, you will:**

- Learn about layout managers
- Use JPanels
- Learn about advanced layout managers
- Understand events and event handling
- Use the AWTEvent class methods
- Use event methods from higher in the inheritance hierarchy
- Handle mouse events

You have been developing Java applets and applications at Event Handlers Incorporated for several months now. "I love this job!" you exclaim to Lynn Greenbrier one day. "I've learned so much, yet there's so much more I don't know. Sometimes I look at a Swing applet that I've created and think how far I have come; other times I realize I barely have a start in the Java programming language."

"Go on," Lynn urges, "what do you need to know more about right now?"

"Well, I wish it were easier to place components accurately within applets and frames," you say. "I'd like to be able to create more-complex applets and applications, and one thing I'm really confused about is handling events. You've taught me about 'registering objects as listeners,' 'listening,' and 'handling,' but I'd like to learn more about the big picture."

"Event handling is a complicated system," Lynn says. "Let's see if I can help you organize it in your mind. After all, we are the Event Handlers!"

## PREVIEWING THE CHAP14 SWING APPLET

Event Handlers Incorporated is developing an applet that the user can manipulate to uncover an advertising slogan. The user passes the mouse over different regions of the applet surface, individually revealing three colored panels. The user can reveal one-third of the advertising slogan at a time by clicking one of the colored panels. The user can also reposition each slogan segment within its panel area by clicking the mouse in a new position. In this chapter, you will learn the techniques used to create this Swing applet. Next you will run the finished version of the Swing applet that is saved on your Student Disk.

**To run the Chap14 Swing applet:**

1. At the command line for the Chapter.14 folder on your Student Disk, type **`appletviewer TestChap14JPanelApplet.html`**. After the Swing applet appears in the Applet Viewer window, move the mouse pointer inside the boundaries of the Swing applet.
2. When you move the mouse pointer inside the Swing applet boundaries, three colored panels are revealed. When your pointer leaves the Swing applet, the background color of the panels changes to black. When your pointer reenters the Swing applet area, the panels' background colors change back to their original colors.
3. Click any one of the three colored panels. Depending on the exact position of your mouse, you will see all or part of a slogan segment. Click each of the remaining colored panels to reveal all three messages, which are shown in Figure 14-1.

Tip

You can examine the code used to create the applet by opening the Chap14JPanelApplet.java file in your text editor.

4. Close the Applet Viewer window.

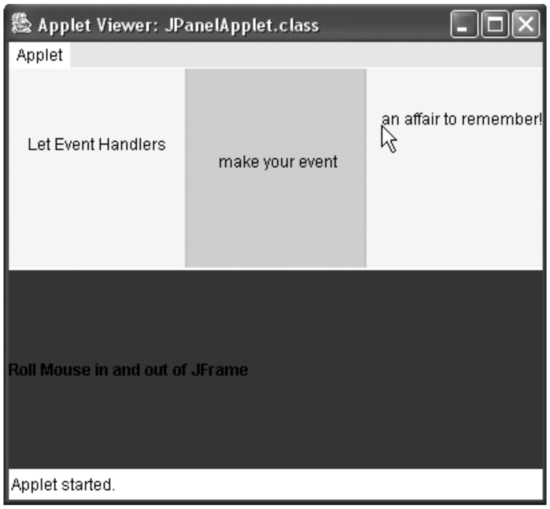


**Figure 14-1** Chap14JPanelApplet applet with three messages

## Learning About Layout Managers

When you add more than one or two components to a JFrame, Swing applet, or any other container, you can spend a lot of time computing exactly where to place each component so that the layout is attractive and no component obscures another one. Another alternative is to use a layout manager. A **layout manager** is an interface class that is part of the JDK. The layout manager aligns your components so they neither crowd each other nor overlap. For example, one layout manager arranges components in equally spaced columns and rows; another layout manager centers components within their container.

Each layout manager defines methods that arrange components within a container, and each component you place within a Container can also be a container itself, so you can assign layout managers within layout managers. The Java platform supplies layout managers that range from the very simple—FlowLayout and GridLayout, to the special purpose—BorderLayout and CardLayout, to the very flexible—GridBagLayout and BoxLayout. Table 14-1 shows each layout manager and typical situations where they are commonly used.

**Table 14-1** Swing layout managers

| Swing layout manager | When to use |
|---|---|
| BorderLayout | Use when you add components to a maximum of five sections arranged in North, South, East, West, and Center positions. |
| FlowLayout | Use when you need to add components from left to right; FlowLayout automatically moves to the next row when needed. |
| GridLayout | Use when you need to add components into a grid of rows and columns. |
| CardLayout | Use when you need to add components that are displayed one at a time. |
| BoxLayout | Use when you need to add components into a single row or a single column. |
| GridBagLayout | Use when you need to set size, placement, and alignment constraints for every component that you add. |

## BorderLayout

The **BorderLayout manager** is the default manager for all content panes. You can use the BorderLayout class with any container that has five or fewer components. However, you should be aware that any of the components could be a container that holds even more components. The components fill the screen in five regions named North, South, East, West, and Center. Figure 14-2 shows five JButton objects filling the five regions in an applet.

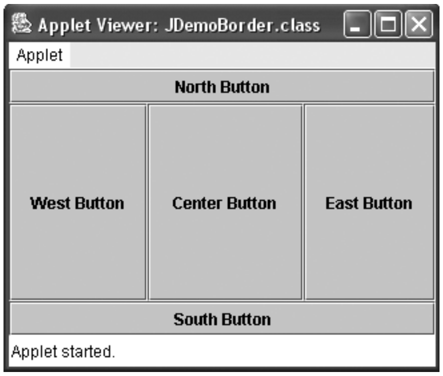


**Figure 14-2** Positions using BorderLayout

When you place exactly five components in a container and use BorderLayout, each component fills one entire region, as illustrated in Figure 14-2. When the program runs, the compiler determines the exact size of each component based on the component's contents. When you resize a container that uses BorderLayout, the regions also change in size. If you drag the container's border to make the container wider, then the North, South, and Center regions become wider, but the East and West regions do not change. If you increase the container's height, then the East, West, and Center regions become taller, but the North and South regions do not change.

When you create a container, you can set its layout manager to BorderLayout with the statement `setLayout(new BorderLayout());`. For example, if you create a JFrame with the code `JFrame aFrame = new JFrame();`, then you can set the aFrame's layout manager with `aFrame.setLayout(new BorderLayout());`.

Tip

You may choose to create a BorderLayout manager, but it's not necessary to do so. The program automatically defaults to BorderLayout because it is the default layout manager for all content panes.

When you use the add() method to add a component to a container that uses BorderLayout, you use one of the five area names to specify the region of the container where the component should be placed. For example, when you place a JButton into a container, `add(someButton, "South");` places the someButton object in the South region of the current container, and when you place a JCheckBox into a container, `aNiceFrame.add(someCheckbox, "East");` adds someCheckbox to the East region of aNiceFrame.

When you use BorderLayout with a container, you are not required to add five components. If you add fewer components, any empty component regions disappear and the remaining components expand to fill the available space. Next, so you can observe the effect, you will remove a component from a container that uses BorderLayout.

**To create a Container that uses BorderLayout with only four objects:**

1. Open a new file in your text editor, and then type the following first few lines of a Swing applet that will demonstrate BorderLayout with only four objects:

```
import javax.swing.*;
import java.awt.*;
public class JDemoBorderNoNorth extends JApplet
{
```

2. Enter the following lines to create four buttons. Note that the North button is purposefully omitted:

```
    private JButton sb = new JButton("South Button");
    private JButton eb = new JButton("East Button");
    private JButton wb = new JButton("West Button");
    private JButton cb = new JButton("Center Button");
```

3. Enter the following code to create the init() method in which you will set the layout manager to BorderLayout:

```
    public void init()
    {
      Container con = getContentPane();
      con.setLayout(new BorderLayout());
```

4. Next enter the following code to add the four buttons to the four regions, along with the closing curly braces for the init() method and class:

```
          con.add(sb,"South");
          con.add(eb,"East");
          con.add(wb,"West");
          con.add(cb,"Center");
       }
    }
```

5. Save the program as **JDemoBorderNoNorth.java**, and then compile it using the `javac` command.
6. Open a new text file in your text editor, and then create the following HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JDemoBorderNoNorth.class" WIDTH = 375
   HEIGHT = 300>
</APPLET>
</HTML>
```

7. Save the file as **TestJBorder.html**. Run the Swing applet using the `appletviewer` command. The output looks like Figure 14-3. Notice that there are only four components in this Swing applet and that none has been assigned to the North region. The four components simply expand their sizes to fit the space.

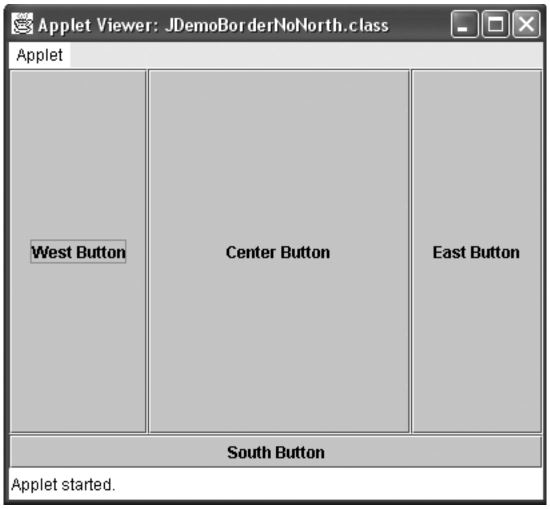


**Figure 14-3** JDemoBorderNoNorth applet

8. Close the Applet Viewer window.

9. Open the **JDemoBorderNoNorth.java** file, and then experiment with removing different components. Run the applet by using the **`appletviewer`** command and observe the results.

## FlowLayout

Remember from Chapter 9 that you can use the FlowLayout class to arrange components in rows across the width of a container. Each component that you add is placed to the right of previously added components. When you use BorderLayout, the components you add fill their regions. However, when you use FlowLayout, each component retains its default size; for example, a JButton will be large enough to hold its text. When you use BorderLayout, if you resize the window, the components change size accordingly. With FlowLayout, when you resize the window, each component retains its size, but it might become partially obscured or change position.

Next you will modify a BorderLayout Swing applet to demonstrate FlowLayout.

**To demonstrate FlowLayout:**

1. Open the **JDemoBorderNoNorth.java** file in your text editor, and immediately save it as **JDemoFlowRight.java**. Position the insertion point at the end of the opening curly brace for the JDemoBorderNoNorth.class and press **[Enter]**. Create a new North JButton with the statement **`private JButton nb = new JButton("North Button");`**.
2. Change the class name from JDemoBorderNoNorth to **`JDemoFlowRight`**.
3. Within the init() method, change the setLayout() statement to use FlowLayout and right align:
   ```
   con.setLayout(new FlowLayout(FlowLayout.RIGHT));
   ```
4. Add the North button with the statement **`con.add(nb);`**. Remove the "**South**", "**East**", "**West**", and "**Center**" locations from the remaining statements.
5. Save the file, and then compile it using the **`javac`** command.
6. Open a new text file, and then create the following HTML document to host the Swing applet:
   ```
   <HTML>
   <APPLET CODE = "JDemoFlowRight.class" WIDTH = 300
      HEIGHT = 150>
   </APPLET>
   </HTML>
   ```
7. Save the HTML document as **TestJDemoFlowRight.html** in the Chapter.14 folder on your Student Disk, and then run the applet using the **`appletviewer`** command. Your output should look like Figure 14-4.

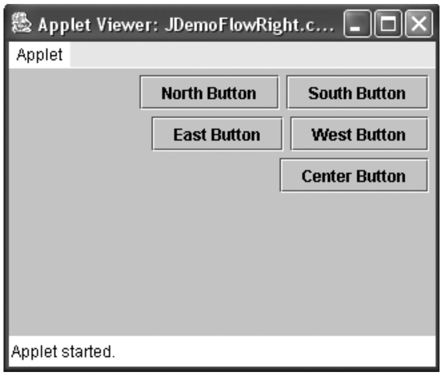


**Figure 14-4** JDemoFlowRight Swing applet

8. Experiment with widening and narrowing the Applet Viewer window, and observe how the components realign themselves.
9. Close the Applet Viewer window.

## GridLayout

If you want to arrange components into equal rows and columns, you can use the GridLayout class. When you create a GridLayout object, you indicate the numbers of rows and columns you want, and then the container surface is divided into a grid, much like the screen you see when using a spreadsheet program. For example, the statement `setLayout(new GridLayout(4,5));` establishes a GridLayout with four horizontal rows and five vertical columns.

Tip

You specify rows first, and then columns, which is the same technique you used when specifying two-dimensional arrays in Chapter 8.

As you add new Components to a GridLayout, they are positioned left-to-right across each row, in sequence. Unfortunately, you can't skip a position or specify an exact position for a component. You can specify zero for either the row or column figure, which will provide an unlimited number of rows or columns. You can also specify a vertical and horizontal gap measured in pixels using two additional arguments. For example, the statement `setLayout(new GridLayout(4,5,2,6));` establishes a GridLayout with four horizontal rows and five vertical columns, a horizontal gap of two pixels, and a vertical gap of six pixels.

Next you will modify a BorderLayout Swing applet to demonstrate GridLayout.

**To demonstrate GridLayout:**

1. Open the **JDemoFlowRight.java** file in your text editor, and then save the file as **JDemoGrid.java**.
2. Change the class name from JDemoFlowRight to **`JDemoGrid`**.
3. Within the init() method, type the following statement to change the setLayout() method call to establish a GridLayout with two rows, three columns, a horizontal space of two pixels, and a vertical space of four pixels:

   ```
   con.setLayout(new GridLayout(2,3,2,4));
   ```
4. Save the file, and then compile it using the **`javac`** command.
5. Open the **TestJBorder.html** file in your text editor, change the Swing applet reference to **`JDemoGrid.class`**, and then save the file as **TestJGrid.html** in the Chapter.14 folder on your Student Disk.
6. Use the **`appletviewer`** command to run the applet, and then compare your output to Figure 14-5. The components are arranged in two rows and three columns. Because there are only five components, one grid position still is available.
7. Close the Applet Viewer window.

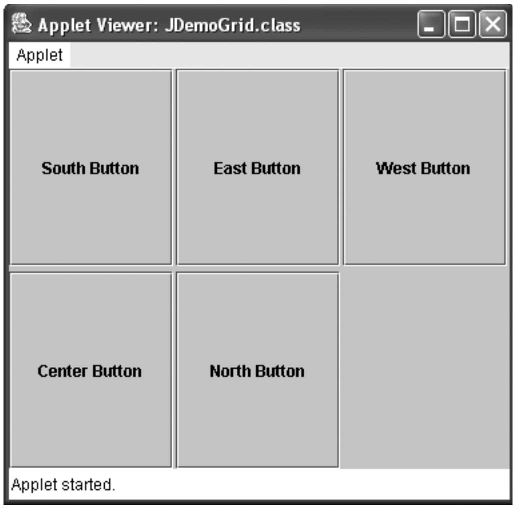


Figure 14-5 JDemoGrid Swing applet

## CardLayout

The **card layout manager** generates a stack of containers or components, one on top of another much like a blackjack dealer reveals cards one at a time from the top of a deck of cards. Each container in the group is referred to as a **card**. A card layout is created from the CardLayout class using one of two constructors:

- CardLayout() creates a new card layout without a horizontal or vertical gap.
- CardLayout(int hgap, int vgap) creates a new card layout with the specified horizontal and vertical gaps. The horizontal gaps are placed at the left and right edges. The vertical gaps are placed at the top and bottom edges.

For example, the statement `CardLayout cl = new CardLayout();` creates the card with no horizontal or vertical gaps, while the statement `CardLayout cl = new CardLayout(5,10);` creates the card with a horizontal gap of 5 pixels and a vertical gap of 10 pixels.

After you set the layout manager, as in the statement `con.setLayout(cl);`, you use a slightly different add() method to add to the layout. The method is add(String, container); Container in this example is the Container con. The following statements have five effects:

```
CardLayout cl = new CardLayout();
Container con = this.getContentPane();
con.setLayout(cl);
JButton button1 = new JButton();
con.add("Options", button1);
```

1. Create a CardLayout object named cl.
2. Create a Container object named con for the program.
3. Set the layout of the Container to CardLayout.
4. Create a JButton called button1.
5. Add the Button to the Container con.

The String "Options" represents the name of the card; you can use any name you want. If you have several cards, you might choose to name them "Option1", "Option2", and so on.

Usually in a program that has a card layout manager, a change of cards is triggered by a user's action. For example, a user could select a card by clicking a button that had been registered as an event listener: `option1.addActionListener(this);`. The statement `next(getContentPane())` flips to the next card of the specified container, and the statement `previous(getContentPane())` flips to the previous card of the specified container. For example, the statement `cl.next(getContentPane());` flips to the next card held in the parent content pane.

Next you will create a CardLayout with five cards, each representing a kind of entertainment provided by Event Handlers Incorporated.

**To demonstrate CardLayout:**

1. Open a new file in your text editor, and then type the following first few lines of a Swing applet that demonstrates CardLayout with five objects:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JCardLayout extends JApplet implements
   ActionListener
{
```

2. Create a new CardLayout manager cl with horizontal and vertical gaps, and declare five buttons as the Components to be displayed in the program:

```
  CardLayout cl = new CardLayout(5,5);
  JButton b1, b2, b3, b4, b5;
```

3. Within the init() method, type the following statement to change the setLayout() method call to establish a CardLayout:

```
  public void init()
  {
   Container con = this.getContentPane();
   con.setLayout(cl);
```

4. Create the JButton b1 with the caption “Clown”. Add b1 to the Container with the name “opt1”, then register b1 as an event listener. Repeat this process to create four additional buttons for “Singer”, “Magician”, “Poet”, and “Lion Tamer”.

```
    b1 = new JButton("Clown");
    con.add("opt1",b1);
    b1.addActionListener(this);
    b2 = new JButton("Singer");
    con.add("opt2",b2);
    b2.addActionListener(this);
    b3 = new JButton("Magician");
    con.add("opt3",b3);
    b3.addActionListener(this);
    b4 = new JButton("Poet");
    con.add("opt4",b4);
    b4.addActionListener(this);
    b5 = new JButton("Lion Tamer");
    con.add("opt5",b5);
    b5.addActionListener(this);
}
```

5. Add the following ActionPerformed method to flip from one card to the next. The cards will display in the order in which they were placed in the content pane.

```
       public void actionPerformed(ActionEvent e)
       {
         cl.next(getContentPane());
       }
     }
   }
```

6. Save the file as **JCardLayout.java**, and then compile it using the **`javac`** command.
7. Open the **TestJGrid.html** file in your text editor, change the Swing applet reference to **`JCardLayout.class`**, and then save the file as **TestJCardLayout.html** in the Chapter.14 folder on your Student Disk.
8. Use the **`appletviewer`** command to run the Swing applet, and then compare your output to Figure 14-6. Note that the first card that appears contains the name of an entertainment act. Click a card and the next card appears with a new name. When you click the last card, the first card appears again.
9. Close the Applet Viewer window.

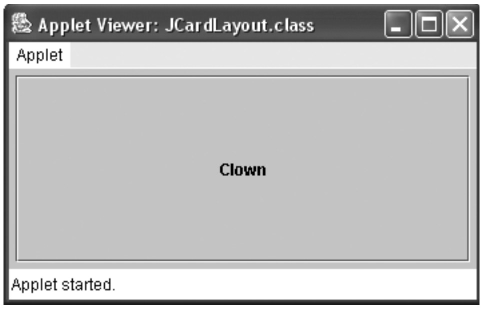


**Figure 14-6** JCardLayout Swing applet

## USING JPANELS

Using the BorderLayout, FlowLayout, GridLayout, and CardLayout managers provides a limited number of screen arrangements. You can increase the number of possible component arrangements by using the JPanel class. A JPanel is similar to a JWindow in that a JPanel is a surface on which you can place components. But a JPanel is *not* a JWindow; it is a sibling of a JComponent, as shown in Figure 14-7. A JPanel is a Container, which means that it can contain other components. For example, you can create a Swing applet using BorderLayout and place a JPanel in any of the five regions. Then within the North JPanel, you can place four JButtons using GridLayout, and within the East JPanel, you can place three JLabels using FlowLayout. By using JPanels within JPanels, you can create an infinite variety of screen layouts.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
```

**Figure 14-7** Structure of the JPanel class

When you create a JPanel object, you can use one of two constructors:

- JPanel() creates a new JPanel with a double buffer and a flow layout.
- JPanel(LayoutManager layout) creates a new buffered JPanel with the specified layout manager.

Next you will create a Swing applet that uses a layout manager and contains a JPanel that uses a different layout manager. To begin, you will create one JPanel named wp that holds JButtons indicating the states in which Event Handlers Incorporated does business. Using GridLayout, you will place three JButtons and a JLabel in this JPanel to represent three states. When the user clicks a JButton representing Wyoming, for example, the Swing applet displays the locations of Event Handlers offices in that state. For simplicity, you will activate only one of the three JButtons.

**To create the JWesternPanel object:**

1. Open a new file in your text editor, and then enter the following first few lines of the JWesternPanel class. The JWesternPanel class extends JApplet and implements ActionListener because the JPanel contains a clickable JButton. Both the JButton and the JLabel are placed at the beginning so that their scope extends throughout the JWesternPanel class.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JWesternPanel extends JApplet
  implements ActionListener
{
  JButton wyButton = null;
  JLabel infoLabel = null;
```

2. Create an init() method for the Swing applet, and enter the following code to create a Container for the Swing applet and set the Container's layout to BorderLayout:

```
    public void init()
    {
      Container container = this.getContentPane();
      container.setLayout(new BorderLayout());
```

3. Create a new JPanel with a GridLayout using two rows and two columns to hold the JLabel and JButtons. Create four Components: three JButtons for the three Western states in which Event Handlers operates, and a JLabel.

```
        JPanel wp = new JPanel(new GridLayout(2,2,2,2));
        wyButton = new JButton("Wyoming");
        JButton coButton = new JButton("Colorado");
        JButton nvButton = new JButton("Nevada");
        infoLabel = new JLabel(" Location Info ");
```

4. Add the Wyoming JButton to the grid layout of the JPanel. Register the JButton as an ActionListener so users can click the Wyoming Button. Add the other JButtons and the JLabel to the grid layout. For now, these components are not clickable, so don't use the addActionListener() method with them.

```
        wyButton = new JButton("Wyoming");
        wp.add(wyButton);
        wyButton.addActionListener(this);
        wp.add(coButton);
        wp.add(nvButton);
        wp.add(infoLabel);
```

5. Create four new JButtons representing the North, South, East, and Center regions of the Swing applet Container. Add these JButtons along with the JPanel, placing the JPanel in the West region.

```
        JButton nb = new JButton("North Button");
        JButton sb = new JButton("South Button");
        JButton eb = new JButton("East Button");
        JButton cb = new JButton("Center Button");
        container.add(nb,"North");
        container.add(sb,"South");
        container.add(eb,"East");
        container.add(wp,"West");
        container.add(cb,"Center");
    }
```

6. When the user clicks the Wyoming Button (which is the only active JButton), the following actionPerformed() method executes. This method displays the name of the Event Handlers Incorporated Wyoming location—Cody. After the method, add a closing curly brace for the class.

```
    public void actionPerformed(ActionEvent e)
    {
      Object source = e.getSource();
      if (source == wyButton)
      {
         infoLabel.setText("Cody");
      }
    }
  }
```

7. Save the file as **JWesternPanel.java** in the Chapter.14 folder on your Student Disk, and then compile the file using the **`javac`** command.
8. Open a new text file, and then create the following HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JWesternPanel.class" WIDTH = 400
   HEIGHT = 300>
</APPLET>
</HTML>
```

9. Save the HTML document as **TestJWesternPanel.html** in the Chapter.14 folder on your Student Disk, and then run the Swing applet using the **`appletviewer`** command. Your output should look like Figure 14-8.

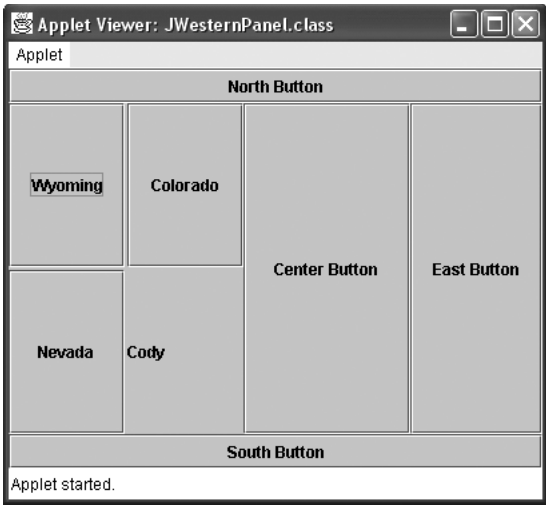


**Figure 14-8** JWesternPanel Swing applet

10. Click the **North**, **South**, **Center**, and **East** JButtons. Nothing happens, because you have not activated these JButtons. Similarly, in the West region, click the **Colorado** and **Nevada** JButtons; again, no actions result. Now, click the **Wyoming** JButton; the JLabel within the JWestern Panel displays Cody, the city name of the Wyoming Event Handlers location.
11. Close the Applet Viewer window.

## Learning About Advanced Layout Managers

Like Components, professional Java programmers are constantly creating new layout managers. You are certain to encounter new and interesting layout managers during your programming career; you might even create your own. Information about more-complicated layout managers is available in the JDK documentation, which you can access at the *java.sun.com* Web site.

When GridLayout is not sophisticated enough for your purposes, you can use GridBagLayout. The GridBagLayout class allows you to add components to precise locations within the grid, as well as to indicate that specific Components should span multiple rows or columns within the grid. For example, if you want to create a JPanel with six JButtons, in which two of the JButtons are twice as wide as the others, you can use GridBagLayout. This class is difficult to use because you must set the position and size for each component.

Another layout manager option is the BoxLayout manager of the BoxLayout class. When you use the BoxLayout manager, components are arranged in either a single row or a single column. When a row or column is filled, additional components do not spill over onto another row or column. Instead, the box layout manager tries to make all the components the same height(row) or width(column). When used in conjunction with a JPanel, a row or column of JCheckBoxes or JButtons can be placed in a BoxLayout and then added to the JPanel.

The BoxLayout constructor requires two arguments: the first refers to the container to which the layout manager applies, and the second is a constant value. The constant value can be either `BoxLayout.X_Axis` for a row arrangement, or `BoxLayout.Y_Axis` for a column arrangement.

If you use a **`null`** layout manager, as in **`setLayout(null);`**, then you must use the component class methods setBounds(), setSize(), and setLocation() to position your components. You would do this if you need a complicated layout design that you cannot achieve by using the existing layout managers, or if you want to create a new layout manager.

## Understanding Events and Event Handling

You have already worked with many events in the programs you have written. Beginning in Chapter 9, you learned how to create Swing applets which contain widgets that are controlled by user-initiated events. Now that you understand inheritance and abstract classes, you can take a deeper look at event handling.

Like all Java classes, events are Objects. Specifically, events are Objects that the user initiates, such as key presses and mouse clicks. Many events that occur have significance only for specific components within a program. For example, you have written programs, as well as used programs written by others, in which pressing [Enter] or double-clicking a specific component has no effect. Other events have meaning outside your program; for example, clicking the Close button in the AppletViewer window sends a message to your computer's operating system, which closes the window and stops the program.

The parent class for all event objects is named EventObject, which descends from the Object class. EventObject is the parent of AWTEvent, which in turn is the parent to specific event classes such as ActionEvent and ComponentEvent. Figure 14-9 illustrates the structure of these relationships.

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--java.awt.AWTEvent
              |
              +--java.awt.event.ActionEvent
              +--java.awt.event.AdjustmentEvent
              +--java.awt.event.ItemEvent
              +--java.awt.event.TextEvent
              +--java.awt.event.ComponentEvent
                    |
                    +--java.awt.event.ContainerEvent
                    +--java.awt.event.FocusEvent
                    +--java.awt.event.PaintEvent
                    +--java.awt.event.WindowEvent
                    +--java.awt.event.InputEvent
                          |
                          +--java.awt.event.KeyEvent
                          +--java.awt.event.MouseEvent
```

**Figure 14-9** Relationships among event classes

Tip

The abstract class AWTEvent is contained in the package java.awt.event.

You can see in Figure 14-9 that ComponentEvent is itself a parent to several event classes, including InputEvent, which is parent to KeyEvent and MouseEvent. The family tree for events has roots that go fairly deep, but the class names are straightforward and they share basic roles within your programs. For example, ActionEvents pertain to components that users can click, such as JButtons and JCheckboxes, and TextEvents pertain to components into which the user enters text, such as a JTextField. MouseEvents include determining the location of the mouse and distinguishing between a single- and double-click. Table 14-2 lists some common user actions and the events that are generated from them.

Tip

Because ActionEvents involve the mouse, it is easy to confuse ActionEvents and MouseEvents. If you are interested in ActionEvents, you are interested in changes in a component; if you are interested in MouseEvents, your focus is centered on what the user has done manually with the mouse equipment.

**Table 14-2** Examples of user actions and their resulting event types

| User Action | Resulting Event Type |
|---|---|
| Click a button | ActionEvent |
| Click a component | MouseEvent |
| Click an item in a choice | ItemEvent |
| Click an item in a check box | ItemEvent |
| Change text in a text field | TextEvent |
| Open a window | WindowEvent |
| Iconify a window | WindowEvent |
| Press a key | KeyEvent |

When you write programs with GUI interfaces, you are always handling events that originate with the mouse or keys on specific Components or Containers. Just as your telephone notifies you when you have a call, the computer's operating system notifies you, the user, when an AWTEvent occurs, for example, when the mouse is clicked. Just as you can ignore your phone when you're not expecting or interested in a call, you can ignore AWTEvents. If you don't care about an event, such as when your program contains a component which, when clicked, produces no effect, you simply don't look for a message to occur.

Tip

There is no class named Event; the general event class is AWTEvent.

When you care about events—that is, when you want to listen for an event—you can implement an appropriate interface for your class. Each event class shown in Table 14-2 has a listener interface associated with it so that for every event class, such as <name>Event, there is a similarly named <name>Listener interface.

Tip

Remember that an interface contains only abstract methods, therefore all interface methods are empty. If you implement a listener, you must provide your own methods for all the methods that are part of the interface. Of course, you may leave the methods empty in your implementation.

Tip

Every <name>Event class has a <name>Listener. The MouseEvent class has an additional listener, the MouseMotionListener.

Every <name>Listener interface method has return type `void`, and each takes one argument—an object that is an instance of the corresponding <name>Event class. Thus, the ActionListener interface has a method named actionPerformed(), and its header is `void actionPerformed(ActionEvent e)`. When an action takes place, the actionPerformed() method executes, and e represents an instance of that event. Interface methods, such as actionPerformed(), that are called automatically when an appropriate event occurs, are called **event handlers**.

Tip

If a listener has only one method, there is no need for an adapter. For example, the ActionListener class has one method, actionPerformed(), so there is no ActionAdapter class.

Whether you use a listener or an adapter, you create an event handler when you write code for the listener methods; that is, you tell your class how to handle the event. After you create the handler, you must also register an instance of the class with the component that you want the event to affect. For any <name>Listener, you must use the form object.add<name>Listener(Component) to register an object with the Component that will listen for objects emanating from it. The add<name>Listener() methods, such as addActionListener() and addItemListener(), all work the same way. They register a listener with a component, return `void`, and take a <name>Listener object as an argument. For example, if a Swing applet is an ActionListener and contains a JButton named pushMe, then within the Swing applet, `pushMe.addActionListener(this);` registers this particular applet as a listener for the pushMe JButton. Table 14-3 lists the events with their listeners and handlers.

**Table 14-3** Events, their listeners, and their handlers

| Event | Listener | Handlers |
|---|---|---|
| ActionEvent | ActionListeneraction | Performed(ActionEvent) |
| ItemEvent | ItemListener | itemStateChanged(ItemEvent) |
| TextEvent | TextListener | textValueChanged(TextEvent) |
| AdjustmentEvent | AdjustmentListener | adjustmentValueChanged(AdjustmentEvent) |
| ContainerEvent | ContainerListener | componentAdded(ContainerEvent)<br>componentRemoved(ContainerEvent) |
| ComponentEvent | ComponentListener | componentMoved(ComponentEvent)<br>componentHidden(ComponentEvent)<br>componentResized(ComponentEvent)<br>componentShown(ComponentEvent) |
| FocusEvent | FocusListener | focusGained(FocusEvent)<br>focusLost(FocusEvent) |
| MouseEvent | MouseListenermouse<br><br><br><br><br>MouseMotionListener | Pressed(MouseEvent)<br>mouseReleased(MouseEvent)<br>mouseEntered(MouseEvent)<br>mouseExited(MouseEvent)<br>mouseClicked(MouseEvent)<br>mouseDragged(MouseEvent)<br>mouseMoved(mouseEvent) |

**Table 14-3** Events, their listeners, and their handlers (continued)

| Event | Listener | Handlers |
|---|---|---|
| KeyEvent | KeyListener | keyPressed(KeyEvent)<br>keyTyped(KeyEvent)<br>keyReleased(KeyEvent) |
| WindowEvent | WindowListener | windowActivated(WindowEvent)<br>windowClosing(WindowEvent)<br>windowClosed(WindowEvent)<br>windowDeiconified(WindowEvent)<br>windowIconified(WindowEvent)<br>windowOpened(WindowEvent) |

Next you will create a class that implements KeyListener. You use the **KeyListener interface** when you are interested in actions the user initiates from the keyboard. The KeyListener interface contains three methods—keyPressed(), keyTyped(), and keyReleased(). For most keyboard applications in which the user must type a keyboard key, it is probably not important whether you take resulting action when a user first presses a key, during the key press, or upon the key's release; most likely these events occur in quick sequence. However, on those occasions when you don't want to take action while the user holds down the key, you can place the actions in the keyReleased() method.

It is best to use the keyTyped() method when you want to discover what character was typed. When the user presses a key that does not generate a character (sometimes called **action keys**), such as a function key, then keyTyped() does not execute. The methods keyPressed() and keyReleased() provide the only ways to get information about keys that don't generate characters.

Tip

Java programmers call keyTyped() events "higher level" events because they do not depend on the platform or keyboard layout. In contrast, keyPressed() and keyReleased() events are "lower level" events and do depend on the platform and keyboard layout.

**To create a class that implements KeyListener:**

1. Open a new file in your text editor, and then enter the following first few lines for the JKeyFrame class that implements KeyListener:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JKeyFrame extends JFrame
 implements KeyListener
{
```

2. Create a `null` container to hold the JFrame components. Create a BorderLayout manager for the component layout. Create a JLabel and a JTextField.

   ```
   Container con = null;
   BorderLayout border = new BorderLayout();
   JLabel label = new JLabel("Key Typed:");
   JTextField textField = new JTextField(25);
   ```

3. In the JKeyFrame constructor method, set the JFrame title to JKeyFrame and the default close operation to EXIT_ON_CLOSE. Create the layout object con by calling the getContentPane() method. Set the layout manager to border layout. Add the JLabel and JTextField to the North and South regions of the border layout. Add an ActionListener for each JButton using the keyword `this` to represent the JFrame.

   ```
   public JKeyFrame()
   {
    setTitle("JKey Frame");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    con = this.getContentPane();
    con.setLayout(border);
    con.add(textField,"North");
    con.add(label,"South");
    addKeyListener(this);
    textField.addKeyListener(this);
   }
   ```

4. The following keyTyped() method is one of the three abstract methods contained in KeyListener. To prove that this method is activated when the user presses a key, send the simple message "pressed" to the command line. Use the getKeyChar() method to retrieve the keyboard character typed, and then display the character with the setText() method.

   ```
   public void keyTyped(KeyEvent e)
    {
     System.out.println("typed");
     char c = e.getKeyChar();
     label.setText ("Key Typed: " + c);
    }
   ```

5. Similarly, implement the following keyPressed() method to print "pressed" at the command prompt:

   ```
   public void keyPressed(KeyEvent e)
    {
     System.out.println("pressed");
    }
   ```

6. Implement the following keyReleased() method so it prints "released":

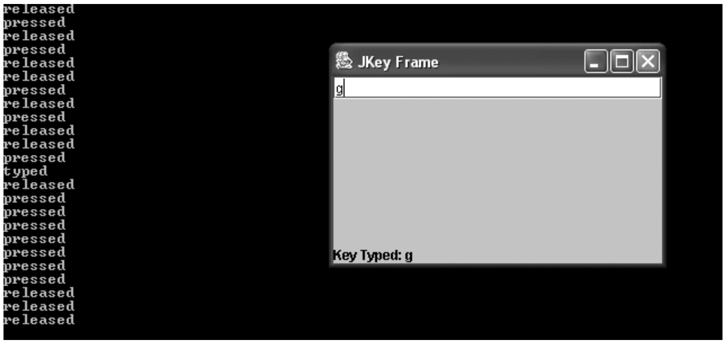
```
public void keyReleased(KeyEvent e)
{
 System.out.println("released");
}
```

7. Add the following main() method that creates a new JFrame named kFrame, sizes it using the setSize() method, and sets its visible property to `true`.

   Remember to add the closing curly brace for the class.

```
  public static void main(String[] arguments)
  {
    JFrame kFrame = new JKeyFrame();
    kFrame.setSize(250,100);
    kFrame.setVisible(true);
  }
 }
```

8. Save the file as **JKeyFrame.java** in the Chapter.14 folder on your Student Disk. Compile the file using the `javac` command, and then run the program using the `java JKeyFrame` command. When the JFrame appears on your screen, notice that it contains a text field in which you can enter data. When you press an alphabetic keyboard key, the character appears in the text field and in the label, and the command line displays three messages: "pressed", "typed", and "released". Figure 14-10 shows these messages. If you press a key, such as F1 or Alt, that does not generate a character, you see "pressed" and "released", but not "typed". If you hold down a key, such as the Alt key, you can generate several "pressed" messages before receiving the "released" message.



```
released
pressed
released
pressed
released
released
pressed
released
pressed
released
released
pressed
typed
released
pressed
pressed
pressed
pressed
pressed
pressed
pressed
released
released
released
```

**Figure 14-10** Output of the JKeyFrame program

## Using AWTEvent Class Methods

In addition to the handler methods included with the event listener interfaces, the AWTEvent classes themselves contain methods. You use many of these methods to determine the nature of and the facts about an event in question. For example, the ComponentEvent class contains a getComponent() method that returns the Component involved in the event. You use the getComponent() method when you create an application with several components, and then the getComponent() method allows you to determine which Component is generating the event. The WindowEvent class contains a similar method, getWindow(), that returns the Window that was the source of the event. Table 14-4 lists some useful methods for many of the event classes.

Tip

All Components have the methods addComponentListener(), addFocusListener(), addMouseListener(), and addMouseMotionListener().

**Table 14-4** Useful Event class methods

| Class | Method | Purpose |
|---|---|---|
| EventObject | `Object getSource()` | Returns the Object involved in the event |
| ComponentEvent | `Component getComponent()` | Returns the Component involved in the event |
| WindowEvent | `Window getWindow()` | Returns the Window involved in the event |
| ItemEvent | `Object getItem()` | Returns the Object that was selected or deselected |
| ItemEvent | `int getStateChange()` | Returns an integer named ItemEvent.SELECTED or ItemEvent.DESELECTED |
| InputEvent | `int getModifiers()` | Returns an integer to indicate which mouse button was clicked |
| InputEvent | `int getWhen()` | Returns a time indicating when the event occurred |
| InputEvent | `boolean isAltDown()` | Returns whether [Alt] was down when the event occurred |
| InputEvent | `boolean isControlDown()` | Returns whether the Ctrl key was down when the event occurred |
| InputEvent | `boolean isShiftDown()` | Returns whether the Shift key was down when the event occurred |
| KeyEvent | `int getKeyChar()` | Returns the Unicode character entered from the keyboard |
| MouseEvent | `int getClickCount()` | Returns the number of mouse clicks; lets you identify the user's double-clicks |
| MouseEvent | `int getX()` | Returns the x-coordinate of the mouse pointer |
| MouseEvent | `int getY()` | Returns the y-coordinate of the mouse pointer |
| MouseEvent | `Point getPoint()` | Returns the Point Object that contains x- and y-coordinates of the mouse location |

You can call any of the methods listed in Table 14-4 by using the object-dot-method format that you use with all class methods. For example, if you have an InputEvent named inEv, and an integer named modInt, then the statement `modInt = inEv.getModifiers();` is valid. You use the getModifiers() method with an InputEvent object, and you can assign the return value to an integer variable. Thus, when you use any of the handler methods from Table 14-3, such as actionPerformed() or itemStateChanged(), they provide you with an appropriate event object. You can use the event object within the handler method to obtain information; you simply add a dot and the desired method name from Table 14-3.

## USING EVENT METHODS FROM HIGHER IN THE INHERITANCE HIERARCHY

When you use an event such as KeyEvent, you can use any of the event's methods. Through the power of inheritance, you can also use methods that belong to any class that is a superclass of the event with which you are working. For example, any KeyEvent has access to the InputEvent, ComponentEvent, AWTEvent, EventObject, and Object methods, as well as the KeyEvent methods.

Next you will use an EventObject method with an ActionEvent. You can accomplish this because every ActionEvent is a descendant of EventObject. Therefore, when you create a Component with several JButton objects, you can use ObjectEvent's getSource() method to determine the source of the ActionEvent.

**To write a class that uses the EventObject method getSource() with an ActionEvent:**

1. Open a new file in your text editor, and then type the following first few lines of the JButtonFrame class that implements the ActionListener interface to respond to JButton clicks:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JButtonFrame extends JFrame
 implements ActionListener
{
```

2. Create the following three JButtons from which the user can choose to change the Frame's background color. Create a `null` container to hold the JFrame components and a container to hold the FlowLayout manager for the component layout.

```
  JButton redButton = new JButton("Red");
  JButton blueButton = new JButton("Blue");
  JButton greenButton = new JButton("Green");
  Container con = null;
  FlowLayout flow = new FlowLayout();
```

3. Begin writing the JButtonFrame constructor and set the layout manager to FlowLayout, the JButtonFrame title to JButtonFrame, and the default close operation to EXIT_ON_CLOSE. Create the layout object con by calling the getContentPane(), and then add the three JButtons to the container.

```
public JButtonFrame()
{
 setTitle("JButton Frame");
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 con = this.getContentPane();
 con.setLayout(flow);
 con.add(redButton);
 con.add(blueButton);
 con.add(greenButton);
```

Tip

Earlier in this chapter, you learned that the default layout for a JFrame is BorderLayout.

4. Continue entering the JButtonFrame constructor by setting the background and foreground of the container. Register the JButtonFrame as an ActionListener for each of the three JButton objects, and then close the constructor method.

```
  con.setBackground(Color.white);
  con.setForeground(Color.black);
  redButton.addActionListener(this);
  blueButton.addActionListener(this);
  greenButton.addActionListener(this);
}
```

5. Add the following main() method that creates a new JButtonFrame named bFrame, sizes it using the setSize() method, and sets its visible property to `true`.

```
public static void main(String[] args)
{
 JButtonFrame bFrame = new JButtonFrame();
 bFrame.setSize(350,250);
 bFrame.setVisible(true);
}
```

6. Because JButtonFrame implements ActionListener, you are required to write code for ActionListener's only method, actionPerformed(). The actionPerformed() method provides you with an ActionEvent object with which you can use the EventObject method named getSource() to return the source of the event as an Object class instance. Using the `if...else` structure allows you to compare the

source Object with possible event sources and take the appropriate action. Remember to add the closing curly braces for the method and class.

```
public void actionPerformed(ActionEvent e)
  {
    Object source = e.getSource();
    if(source == redButton)
      con.setBackground(Color.red);
    else if (source == blueButton)
      con.setBackground(Color.blue);
    else if (source == greenButton)
      con.setBackground(Color.green);
  }
}
```

7. Save the file as **JButtonFrame.java** in the Chapter.14 folder on your Student Disk. Compile the file using the **`javac`** command, and then run the program using the **`java JButtonFrame`** command. The output of the program after the Blue button has been clicked is shown in Figure 14-11. Click any of the three color JButtons and note the change in the JFrame's background color. Note that the JFrame listens for action on each of the JButtons, and the single actionPerformed() method executes no matter which JButton is clicked. You achieve different background colors in the JFrame because you use the ObjectEvent method getSource() with the ActionEvent generated by each button click.

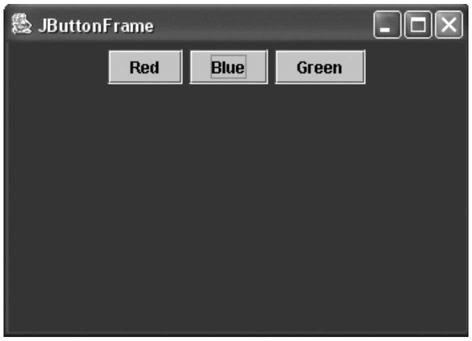


**Figure 14-11** Output of the JButtonFrame program after the Blue button has been clicked

## Handling Mouse Events

Even though Java program users sometimes type characters from a keyboard, when you write GUI programs, you probably expect users to spend most of their time operating a mouse. The MouseMotionListener interface provides you with methods named MouseDragged( ) and MouseMoved( ) that detect the mouse being rolled or dragged across a component surface. The MouseListener interface provides you with methods named mousePressed(), mouseClicked(), and mouseReleased() that are analogous to the keyboard event methods keyPressed(), keyTyped(), and keyReleased(). With a mouse, however, you are interested in more than its key presses; you sometimes simply want to know where a mouse is pointing. The additional interface methods **mouseEntered()** and **mouseExited()** inform you when the user has positioned the mouse over a component (entered) or moves the mouse off a component (exited). To illustrate, you will create a JMouseFrame class that employs these methods. In addition, you will use three MouseEvent methods—getX() and getY(), which return the mouse coordinates, and getClickCount() which helps you distinguish between single- and double-clicks.

**To write the JMouseFrame class:**

1. Open a new file in your text editor, and then type the following first few lines of the JMouseFrame class:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMouseFrame extends JFrame
 implements MouseListener
{
```

2. Create the following `null` container and three integer variables. Two hold the x- and y-positions of the mouse; the third holds the size of a circle you will draw when the mouse is clicked.

```
  Container con = null;
  int x, y;
  int size;
```

3. Enter the following JMouseFrame constructor, which sets the title and adds the MouseListener:

```
  public JMouseFrame()
  {
   setTitle("Mouse Frame");
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   con = this.getContentPane();
   addMouseListener(this);
  }
```

4. Enter the following mousePressed() method to get the x- and y-coordinates from the MouseEvent that initiates the mousePressed() method call. The variables x and y hold the exact position of the mouse location at the time of the event. Also, the mousePressed() method repaints the screen.

```
  public void mousePressed(MouseEvent e)
  {
   x = e.getX();
   y = e.getY();
   repaint();
  }
```

Tip: You learned about the repaint() method in Chapter 9.

5. Enter the following code to set the size variable to 10 or 4, depending on whether the user single- or double-clicks the mouse.

```
public void mouseClicked(MouseEvent e)
{
 if (e.getClickCount() == 2)
  size = 10;
 else size = 4;
  repaint();
}
```

6. Enter the following code to change the JFrame background color to yellow when the user positions the mouse over the JFrame, and then change the background to black when the user places the mouse somewhere else on the screen:

```
public void mouseEntered(MouseEvent e)
{
 con.setBackground(Color.yellow);
}
public void mouseExited(MouseEvent e)
{
 con.setBackground(Color.black);
}
```

7. You don't need any special code for the mouseReleased() method, but you must provide the following method because MouseListener is an abstract interface:

```
public void mouseReleased(MouseEvent e)
{
}
```

8. Recall from Chapter 10 that the Graphics method drawOval() requires four arguments. Envision a rectangle surrounding an oval and provide arguments for the x- and y-coordinates of the upper-left corner and the width and height of the rectangle. The mouseClicked() method sets the size to either 4 or 10, depending on the value returned by getClickCount(). To draw a circle with a diameter of either 8 or 20 pixels, use `x - size` and `y - size` for the upper-left corner of the rectangle, and use `size * 2` for the width and the height. Add the closing curly brace for the method:

```
public void paint(Graphics g)
{
 g.drawOval(x - size, y - size, size * 2, size * 2);
}
```

9. Add the following main() method that creates a new JMouseFrame named mFrame, sizes it using the setSize() method, and sets its visible property to `true`. Add the closing curly brace for the class.

```
  public static void main(String[] args)
  {
   JMouseFrame mFrame = new JMouseFrame();
   mFrame.setSize(250,150);
   mFrame.setVisible(true);
  }
}
```

10. Save the file as **JMouseFrame.java** in the Chapter.14 folder on your Student Disk. Compile the file, and then run the program using the `java JMouseFrame` command. When the JFrame appears on your screen, roll the mouse over the JFrame surface so it turns yellow. Roll your mouse off the JFrame surface and it turns black. Roll the mouse over the JFrame surface and click; a small circle appears at your mouse position. When you click in a new position, the circle relocates. When you double-click in the same position, a larger circle appears in the new mouse position, as shown in Figure 14-12.

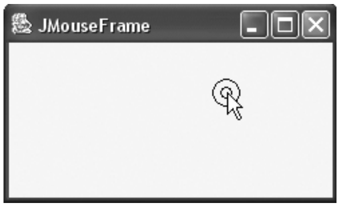


**Figure 14-12** Output of the JMouseFrame program when the mouse is clicked and double-clicked in the same position

## Chapter Summary

- You can use the BorderLayout class with any container that has five or fewer components. The components fill the screen in five regions named North, South, East, West, and Center. The compiler determines the exact size of each component based on the component's contents. When you use BorderLayout with a container and you add fewer than five components, any empty regions disappear.
- You can use the FlowLayout class to arrange Components in rows across the width of a container. When you use FlowLayout, each component retains its default size and automatically is centered within its assigned container.
- You use the GridLayout class to arrange Components into equal rows and columns. As you add new components to a GridLayout, they are positioned left-to-right across each row in sequence.

- The CardLayout class allows you to arrange Components as if they were stacked like index or playing cards. Only one component is visible at a time.
- A JPanel is similar to a JWindow in that a JPanel is a surface on which you can place components. A JPanel is also a container, which means it can contain other components. By using JPanels within other JPanels, you can create an infinite variety of screen layouts.
- The GridBagLayout class allows you to add components to precise locations within the grid, as well as to indicate that specific components should span multiple rows or columns within the grid. The BoxLayout class allows you to add components in a horizontal row or vertical column.
- Events are Objects that the user initiates, such as key presses and mouse clicks. The parent class for all event objects is named EventObject, which descends from the Object class. EventObject is the parent of AWTEvent, which in turn is the parent to specific event classes such as ActionEvent and ComponentEvent.
- When you want to listen for an event, you can implement an appropriate interface for your class, such as ActionListener or WindowListener. The class becomes an event listener. For every event class named <name>Event, there is a listener that is similarly named—<name>Listener. Every <name>Listener interface method has return type `void`, and each takes one argument—an object that is an instance of the corresponding <name>Event class.
- Interface methods that are automatically called when an appropriate event occurs are called event handlers.
- The KeyListener interface contains three methods: keyPressed(), keyTyped(), and keyReleased().
- The MouseListener interface provides you with methods named mousePressed(), mouseClicked(), and mouseReleased(). The additional interface methods mouseEntered() and mouseExited() inform you when the user positions the mouse over a component (entered) or moves the mouse off a component (exited).

## Review Questions

1. If you add fewer than five components to a BorderLayout ______________.
   a. any empty component regions disappear
   b. the remaining components expand to fill the available space
   c. both a and b
   d. none of the above

2. When you resize a Container that uses BorderLayout, ________________.
   a. the Container and the regions both change in size
   b. the Container changes in size, but the regions retain their original sizes
   c. the Container retains its size, but the regions change or might disappear
   d. nothing happens
3. When you create a JFrame named myFrame, you can set its layout manager to BorderLayout with the statement ________________.
   a. `myFrame.setLayout = new BorderLayout();`
   b. `myFrame.setLayout(new BorderLayout());`
   c. `setLayout(myFrame = new BorderLayout());`
   d. `setLayout(BorderLayout(myFrame));`
4. Which is the correct syntax for adding a JButton named b1 to a Container named con when using CardLayout?
   a. `con.add(b1);`
   b. `con.add("b1");`
   c. `con.add("Options", b1);`
   d. none of the above
5. You can use the ________________ class to arrange components in a single row or column of a container.
   a. FlowLayout
   b. BorderLayout
   c. CardLayout
   d. BoxLayout
6. When you use a ________________, the components you add fill their region; they do not retain their default size.
   a. FlowLayout
   b. BorderLayout
   c. FixedLayout
   d. RegionLayout
7. The statement ________________ ensures that components are placed left-to-right across the Swing applet surface until the first row is full, at which point a second row is started at the applet surface's left edge.
   a. `setLayout(FlowLayout.LEFT);`
   b. `setLayout(new FlowLayout(LEFT));`
   c. `setLayout(new FlowLayout(FlowLayout.LEFT));`
   d. `setLayout(FlowLayout(FlowLayout.LEFT));`

8. The GridBagLayout class allows you to ________________.
   a. add components to precise locations within the grid
   b. indicate that specific components should span multiple rows or columns within the grid.
   c. both a and b
   d. none of the above
9. The statement setLayout(new GridLayout(2,7)); establishes a GridLayout with ________________ horizontal row(s).
   a. zero
   b. one
   c. two
   d. seven
10. As you add new components to a GridLayout, ________________.
    a. they are positioned left-to-right across each row in sequence
    b. you can specify exact positions by skipping some positions
    c. both of the above
    d. none of the above
11. A JPanel is a ________________.
    a. Window
    b. Container
    c. both of the above
    d. none of the above
12. The ________________ class allows you to arrange components as if they are stacked like index or playing cards.
    a. GameLayout
    b. CardLayout
    c. BoxLayout
    d. GridBagLayout
13. AWTEvent is the child class of ________________.
    a. EventObject
    b. Event
    c. ComponentEvent
    d. ItemEvent

14. When a user clicks a JPanel, the action generates a(n) ________________.
    a. ActionEvent
    b. MouseEvent
    c. ButtonEvent
    d. none of the above
15. Event handlers are ________________.
    a. abstract classes
    b. concrete classes
    c. listeners
    d. methods
16. The return type of getComponent() is ________________.
    a. Object
    b. Component
    c. int
    d. void
17. The KeyEvent method getKeyChar() returns a(n) ________________.
    a. int
    b. char
    c. KeyEvent
    d. AWTEvent
18. The MouseEvent method that allows you to identify double-clicks is ________________.
    a. getDouble()
    b. isClickDouble()
    c. getDoubleClick()
    d. getClickCount()
19. You can use the ________________ method to determine the Object where an ActionEvent originates.
    a. getObject()
    b. getEvent()
    c. getOrigin()
    d. getSource()

20. The mousePressed() method is originally defined in the ________________.
    a. MouseListener interface
    b. MouseEvent event
    c. MouseObject object
    d. AWTEvent class

## EXERCISES

1. Create a JFrame and set the layout to BorderLayout. Place a JButton containing the name of a politician in each region (left, center, and right, or West, Center, and East). Each politician's physical position should correspond to your opinion of his or her political stance. Save the program as **JPoliticalFrame.java** in the Chapter.14 folder of your Student Disk.
2. Modify the JWesternPanel class you created in this chapter so the user can choose Northern states in addition to Western states. Create the necessary Northern state buttons and activate one of them to display the city location of Event Handlers Incorporated in the Northern region when clicked. Save the program as **JDemoNorth.java** in the Chapter.14 folder of your Student Disk.
3. Use the CardLayout class to write a program that displays a series of cards that make a Royal Flush in hearts (Ace, King, Queen, Jack, and 10). Save the program as **JRoyalFlush.java** in the Chapter.14 folder of your Student Disk.
4. Create 26 JButtons, each labeled with a single, different letter of the alphabet. Create a Swing applet to hold five JPanels in a five-by-one grid. Place six JButtons within the first four JPanels and two JButtons within the fifth JPanel of the Swing applet. Add a JLabel to the fifth JPanel. When the user clicks a JButton, the text of the JLabel is set to "Folder X", where X is the letter of the alphabet that is clicked. Save the program as **JFileCabinet.java** in the Chapter.14 folder of your Student Disk.
5. Create a JFrame that holds four buttons with the names of four different fonts. Draw any String using the font that the user selects. Save the program as **JFontFrame.java** in the chapter.14 folder of your Student Disk.
6. Create a JFrame that uses BorderLayout. Place a JButton in the Center region. Each time the user clicks the JButton, change the background color in one of the regions. Save the program as **JColorFrame.java** in the Chapter.14 folder of your Student Disk.
7. Create a JFrame with JPanels, a JButton, and a JLabel. When the user clicks the JButton, reposition the JLabel to a new location in a different JPanel. Save the program as **JMovingFrame.java** in the Chapter.14 folder of your Student Disk.
8. Write a JFrame application whose appearance and behavior mimics that of the Chap14JPanelApplet Swing applet in this chapter. Save the program as **JFrameApp.java** in the Chapter.14 folder of your Student Disk.

9. Create a class named JPanelOptions that extends JPanel, and whose constructor accepts two colors and a String. Use the colors for background and foreground to display the String. Create a Swing applet named JTeamColors with GridLayout. Display four JPanelOptions JPanels to display the names (in their team colors) of four of your favorite sports teams. Save the program as **JTeamColors.java** in the Chapter.14 folder of your Student Disk.
10. Write a program that lets you determine the integer value returned by the InputEvent method getModifiers() when you press your left, right, or (if you have one) middle mouse button. Save the program as **JLeftOrRight.java** in the Chapter.14 folder of your Student Disk.
11. Write a Swing applet that displays car maintenance services (oil change, tune-up, etc.). Allow the user to select any number of services. If the user clicks the right mouse button, display a message that the user wants service ASAP. Display the choices. Save the program as **JMaintenance.java** in the Chapter.14 folder of your Student Disk.
12. Write a Swing applet that uses a JPanel to show the messages "Mouse Entered" and "Mouse Exited" when the mouse enters and exits the Swing applet. Also, when the mouse is clicked on the Swing applet, a message "Mouse Clicked here" should appear. Save the program as **JMouse.java** in the Chapter.14 folder of your Student Disk.
13. Write a Swing applet to show the messages "Don't Move it! Drag the Mouse!" and "Don't Drag it! Move the Mouse!" when the mouse is alternately rolled and dragged on the Swing applet. Save the program as **JMouseMotion.java** in the Chapter.14 folder of your Student Disk.
14. Each of the following files in the Chapter.14 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugFourteen1.java will become FixDebugFourteen1.java. DebugFourteen2 and DebugFourteen3 are Swing applets. You can use the file TestDebugFourteen.html on your Student Disk to test these Swing applets.
    a. DebugFourteen1.java
    b. DebugFourteen2.java
    c. DebugFourteen3.java
    d. DebugFourteen4.java

## CASE PROJECT

The Programmers Organization wants you to reprogram one of its Java applications so it uses panels and layout managers. The program JExperience is in the Chapter.14 folder of your Student Disk. Although the program compiles and runs, the appearance of the components in the GUI could be greatly enhanced. Using the skills you learned in this chapter, enhance the layout of the program. Save the program as **JNewExperience.java** in the Chapter.14 folder of your Student Disk.