CHAPTER 15

Advanced GUI Topics

In this chapter, you will:

- Use content panes
- Use color
- Learn more about layout managers
- Use `JPanel`s to increase layout options
- Create `JScrollPanes`
- Understand events and event handling more thoroughly
- Use the `AWTEvent` class methods
- Handle mouse events
- Use menus

Understanding the Content Pane

The `JFrame` class is a **top-level container** Java `Swing` class. (The other two top-level container classes are `JDialog` and `JApplet`.) Every GUI component that appears on the screen must be part of a containment hierarchy. A **containment hierarchy** is a tree of components that has a top-level container as its root (that is, at its uppermost level). Every top-level container has a **content pane** that contains all the visible components in the container's user interface. The content pane can directly contain components like `JButtons`, or it can hold other containers, like `JPanels`, that in turn contain such components.

A top-level container can contain a menu bar. A **menu bar** is a horizontal strip that conventionally is placed at the top of a container and contains user options. The menu bar, if there is one, is just above (and separate from) the content pane. A **glass pane** resides above the content pane. Figure 15-1 shows the relationship between a `JFrame` and its root, content, and glass panes.

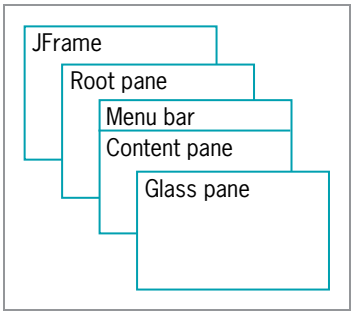

Figure 15-1 Parts of a `JFrame`

The glass pane is a powerful container feature. Tool tips, which you learned about in Chapter 14, reside on the glass pane. You also can draw your own graphics on the glass pane "on top of" components on a `JFrame` or `JApplet`. (You will learn about drawing in the *Graphics* chapter, and about `JApplets` in the chapter *Applets, Images, and Sound*.) If you add a `MouseListener` to the glass pane, it prevents the mouse from triggering events on the components below the glass pane on the `JFrame`.

An additional layered pane exists above the root pane, but it is not often used explicitly by Java programmers. For more details, see the Java Web site.

Whenever you create a `JFrame` (or other top-level container), you can get a reference to its content pane using the **`getContentPane()` method**. In Chapter 14, you added and removed components from `JFrames` and set their layout managers without understanding you were using the content pane. You had this ability because Java automatically converts `add()`,

remove(), and setLayoutManager() statements to more complete versions. For example, the following three statements are equivalent within a class that descends from JFrame:

```
this.getContentPane().add(aButton);
getContentPane().add(aButton);
add(aButton);
```

In the first statement, this refers to the JFrame class in which the statement appears, and getContentPane() provides a reference to the content pane. In the second statement, the this reference is implied. In the third statement, both the this reference and the getContentPane() call are implied.

Although you do not need to worry about the content pane if you only add components to, remove components from, or set the layout manager of a JFrame, you must refer to the content pane for all other actions, such as setting the background color.

When you write an application that adds multiple components to a content pane, it is more efficient to declare an object that represents the content pane than to keep calling the getContentPane() method. For example, consider the following code in a JFrame class that adds three buttons:

```
getContentPane().add(button1);
getContentPane().add(button2);
getContentPane().add(button3);
```

You might prefer to write the following statements. The call to getContentPane() is made once, its reference is stored in a variable, and the reference name is used repeatedly with the call to the add() method:

```
Container con = getContentPane();
con.add(button1);
con.add(button2);
con.add(button3);
```

As an example, the class in Figure 15-2 creates a JFrame like the ones you created throughout Chapter 14, although to keep the example simple, the default close operation was not set and the button was not assigned any tasks.

```
import java.awt.*;
import javax.swing.*;
public class JFrameWithExplicitContentPane extends JFrame
{
   private final int SIZE = 180;
   private Container con = getContentPane();
   private JButton button = new JButton("Press Me");
```

Figure 15-2 The JFrameWithExplicitContentPane class *(continues)*

(*continued*)


```
    public JFrameWithExplicitContentPane()
    {
       super("Frame");
       setSize(SIZE, SIZE);
       con.setLayout(new FlowLayout());
       con.add(button);
    }
    public static void main(String[] args)
    {
       JFrameWithExplicitContentPane frame =
          new JFrameWithExplicitContentPane();
       frame.setVisible(true);
    }
}
```

Figure 15-2 The `JFrameWithExplicitContentPane` class

In Figure 15-2, the `getContentPane()` method assigns a reference to a `Container` named `con`, and the `Container` reference is used later with the `setLayout()` and `add()` methods. Figure 15-3 shows the result. The frame constructed from the class in Figure 15-2 is identical to the one that would be constructed if the shaded parts were omitted.

Figure 15-3 Output of the `JFrameWithExplicitContentPane` application

When you want to use methods other than `add()`, `remove()`, or `setLayout()`, you must use a content pane. In the next sections you will learn about methods such as `setBackground()`, which is used to change a `JFrame`'s background color, and `setLayout()`, which is used to change a `JFrame`'s layout manager. If you use these methods with a `JFrame` instead of its content pane, the user will not see the results.

TWO TRUTHS & A LIE

Understanding the Content Pane

1. Every Java component has a content pane that contains all the visible parts a user sees.
2. Whenever you create a `JFrame`, you can get a reference to its content pane using the `getContentPane()` method.
3. When you change the background color or layout of a `JFrame`, you should change the content pane and not the `JFrame` directly.

The false statement is #1. Every *top-level container* has a content pane that contains all the visible components in the container's user interface.

Using Color

The **`Color` class** defines colors for you to use in your applications. The `Color` class can be used with the `setBackground()` and `setForeground()` methods of the `Component` class to make your applications more attractive and interesting. When you use the `Color` class, you include the statement `import java.awt.Color;` at the top of your class file.

The statement `import java.awt.*` uses the wildcard to import all of the types in the `java.awt` package, but it does not import `java.awt.Color`, `java.awt.Font`, or any other packages within `awt`. If you plan to use the classes from `java.awt` and from `java.awt.Color`, you must use both `import` statements.

The `Color` class defines named constants that represent 13 colors, as shown in Table 15-1. Java constants are usually written in all uppercase letters, as you learned in Chapter 2. However, Java's creators declared two constants for every color in the `Color` class—an uppercase version, such as `BLUE`, and a lowercase version, such as `blue`. Earlier versions of Java contained only the lowercase `Color` constants. (The uppercase `Color` constants use an underscore in `DARK_GRAY` and `LIGHT_GRAY`; the lowercase versions are a single word: `darkgray` and `lightgray`.)

BLACK	GREEN	RED
BLUE	LIGHT_GRAY	WHITE
CYAN	MAGENTA	YELLOW
DARK_GRAY	ORANGE	
GRAY	PINK	

Table 15-1 `Color` class constants

You can also create your own `Color` object with the following statement:

```
Color someColor = new Color(r, g, b);
```

In this statement, `r`, `g`, and `b` are numbers representing the intensities of red, green, and blue you want in your color. The numbers can range from 0 to 255. For example, the color black is created using `r`, `g`, and `b` values 0, 0, 0, and white is created by 255, 255, 255. The following statement produces a dark purple color that has red and blue components, but no green.

```
Color darkPurple = new Color(100, 0, 100);
```

You can create more than 16 million custom colors using this approach. Some computers cannot display each of the 16 million possible colors; each computer displays the closest color it can to the requested color.

You can discover the red, green, or blue components of any existing color with the methods `getRed()`, `getGreen()`, and `getBlue()`. Each of these methods returns an integer. For example, you can discover the amount of red in `MAGENTA` by displaying the value of `Color.MAGENTA.getRed();`.

Figure 15-4 shows a short application that sets the background color of a `JFrame`'s content pane and sets both the foreground and background colors of a `JButton`. Figure 15-5 shows the output.

```
import java.awt.*;
import javax.swing.*;
import java.awt.Color;
public class JFrameWithColor extends JFrame
{
   private final int SIZE = 180;
   private Container con = getContentPane();
   private JButton button =
      new JButton("Press Me");
```

Figure 15-4 The `JFrameWithColor` class *(continues)*

(*continued*)

```
    public JFrameWithColor()
    {
       super("Frame");
       setSize(SIZE, SIZE);
       con.setLayout(new FlowLayout());
       con.add(button);
       con.setBackground(Color.YELLOW);
       button.setBackground(Color.RED);
       button.setForeground(Color.WHITE);
    }
    public static void main(String[] args)
    {
       JFrameWithColor frame =
          new JFrameWithColor();
       frame.setVisible(true);
    }
}
```

Figure 15-4 The `JFrameWithColor` class

Figure 15-5 Execution of the `JFrameWithColor` application

Because this book is printed in only two colors, you can't see the full effect of setting applications' colors in the figures. However, when you work through the "You Do It" exercises later in this chapter, you can observe the effect of color changes on your own monitor.

TWO TRUTHS & A LIE

Using Color

1. The `Color` class can be used with the `setBackground()` and `setForeground()` methods of the `Component` class to make your applications more attractive and interesting.
2. The `Color` class defines named constants that represent 256 colors.
3. You can create your own `Color` object using values that represent the intensities of red, green, and blue you want in your color.

The false statement is #2. The `Color` class defines named constants that represent 13 colors.

Learning More About Layout Managers

A layout manager is an object that controls the size and position (that is, the layout) of components inside a `Container` object. The layout manager that you assign to a `Container` determines how its components are sized and positioned. Layout managers are interface classes that are part of the JDK; they align your components so the components neither crowd each other nor overlap. For example, you have already learned that the `FlowLayout` layout manager positions components in rows from left to right across their container. Other layout managers arrange components in equally spaced columns and rows or center components within their container. 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 15-2 lists each layout manager and situations in which each is commonly used.

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, and each component takes its preferred size.
`GridLayout`	Use when you need to add components into a grid of rows and columns; each component is the same size.
`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.

Table 15-2 Java layout managers

Using `BorderLayout`

The **`BorderLayout` manager** is the default manager class for all content panes. You can use the `BorderLayout` class with any container that has five or fewer components. (However, any of the components could be a container that holds even more components.) When you use the `BorderLayout` manager, the components fill the screen in five regions: north, south, east, west, and center. Figure 15-6 shows a `JFrame` that uses `BorderLayout`; each of the five regions in the content pane contains a `JButton` object with descriptive text.

Figure 15-6 Output of the `JDemoBorderLayout` application

When you add a component to a container that uses `BorderLayout`, the `add()` method uses two arguments: the component and the region to which the component is added. The `BorderLayout` class provides five named constants for the regions—`BorderLayout.NORTH`, `.SOUTH`, `.EAST`, `.WEST`, and `.CENTER`—or you can use the `String`s those constants represent:

"North", "South", "East", "West", or "Center". Figure 15-7 shows the class that creates the output in Figure 15-6.


```
import javax.swing.*;
import java.awt.*;
public class JDemoBorderLayout extends JFrame
{
   private JButton nb = new JButton("North Button");
   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");
   private Container con = getContentPane();
   public JDemoBorderLayout()
   {
      con.setLayout(new BorderLayout());
      con.add(nb, BorderLayout.NORTH);
      con.add(sb, BorderLayout.SOUTH);
      con.add(eb, BorderLayout.EAST);
      con.add(wb, BorderLayout.WEST);
      con.add(cb, BorderLayout.CENTER);
      setSize(400, 150);
   }
   public static void main(String[] args)
   {
      JDemoBorderLayout frame = new JDemoBorderLayout();
      frame.setVisible(true);
   }
}
```

Figure 15-7 The `JDemoBorderLayout` class

When using `BorderLayout`, you can use the constants `PAGE_START`, `PAGE_END`, `LINE_START`, `LINE_END`, and `CENTER` instead of `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`. Rather than using geographical references, these constants correspond to positions as you might picture them on a printed page. Also, if you add the following import statement at the top of your file, you can simply refer to `CENTER` instead of `BorderLayout.CENTER`:

```
import static java.awt.BorderLayout.*;
```

When you place exactly five components in a container and use `BorderLayout`, each component fills one entire region, as illustrated in Figure 15-6. When the application runs, Java 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 it wider, the north, south, and center regions become wider, but the east and west regions do not change. If you increase the `Container`'s height, the east, west, and center regions become taller, but the north and south regions do not change.

In Figure 15-7, an anonymous BorderLayout object is created when the constructor is called from within the setLayout() method. Instead, you could declare a named BorderLayout object and use its identifier in the setLayout() call. However, it's not necessary to use either technique to specify BorderLayout because it is the default layout manager for all content panes; that's why, in many examples in the last chapter, you had to specify FlowLayout to acquire the easier-to-use manager.

When you use BorderLayout, you are not required to add components into each of the five regions. If you add fewer components, any empty component regions disappear, and the remaining components expand to fill the available space. If any or all of the north, south, east, or west areas are left out, the center area spreads into the missing area or areas. However, if the center area is left out, the north, south, east, or west areas do not change. A common mistake when using BorderLayout is to add a Component to a content pane or frame without naming a region. This can result in some of the components not being visible.

Using FlowLayout

Recall from the last chapter, *Introduction to Swing Components*, that you can use the **FlowLayout manager** class to arrange components in rows across the width of a Container. With FlowLayout, each Component that you add is placed to the right of previously added components in a row; or, if the current row is filled, the Component is placed to start a new row.

When you use BorderLayout, the Components you add fill their regions—that is, each Component expands or contracts based on its region's size. However, when you use FlowLayout, each Component retains its default size, or **preferred size**. For example, a JButton's preferred size is the size that is large enough to hold the JButton's text. When you use BorderLayout and then resize the window, the components change size accordingly because their regions change. When you use FlowLayout and then resize the window, each component retains its size, but it might become partially obscured or change position.

The FlowLayout class contains three constants you can use to align Components with a Container:

- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHT

If you do not specify alignment, Components are center-aligned in a FlowLayout Container by default. Figure 15-8 shows an application that uses the FlowLayout.LEFT and FlowLayout.RIGHT constants to reposition JButtons. In this example, a FlowLayout object named layout is used to set the layout of the content pane. When the user clicks a button, the shaded code in the actionPerformed() method changes the alignment to left or right using the FlowLayout class setAlignment() method. Figure 15-9 shows the application when it starts, how the JButton Components are repositioned after the user clicks the "L" button, and how the Components are repositioned after the user clicks the "R" button.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoFlowLayout extends JFrame implements ActionListener
{
   private JButton lb = new JButton("L Button");
   private JButton rb = new JButton("R Button");
   private Container con = getContentPane();
   private FlowLayout layout = new FlowLayout();
   public JDemoFlowLayout()
   {
      con.setLayout(layout);
      con.add(lb);
      con.add(rb);
      lb.addActionListener(this);
      rb.addActionListener(this);
      setSize(500, 100);
   }
   public void actionPerformed(ActionEvent event)
   {
      Object source = event.getSource();
      if(source == lb)
         layout.setAlignment(FlowLayout.LEFT);
      else
         layout.setAlignment(FlowLayout.RIGHT);
      con.invalidate();
      con.validate();
   }
   public static void main(String[] args)
   {
      JDemoFlowLayout frame = new JDemoFlowLayout();
      frame.setVisible(true);
   }
}
```

Figure 15-8 The `JDemoFlowLayout` application

The last statements in the `JDemoFlowLayout` class call `invalidate()` and `validate()`. The `invalidate()` call marks the container (and any of its parents) as needing to be laid out. The `validate()` call causes the components to be rearranged based on the newly assigned layout.

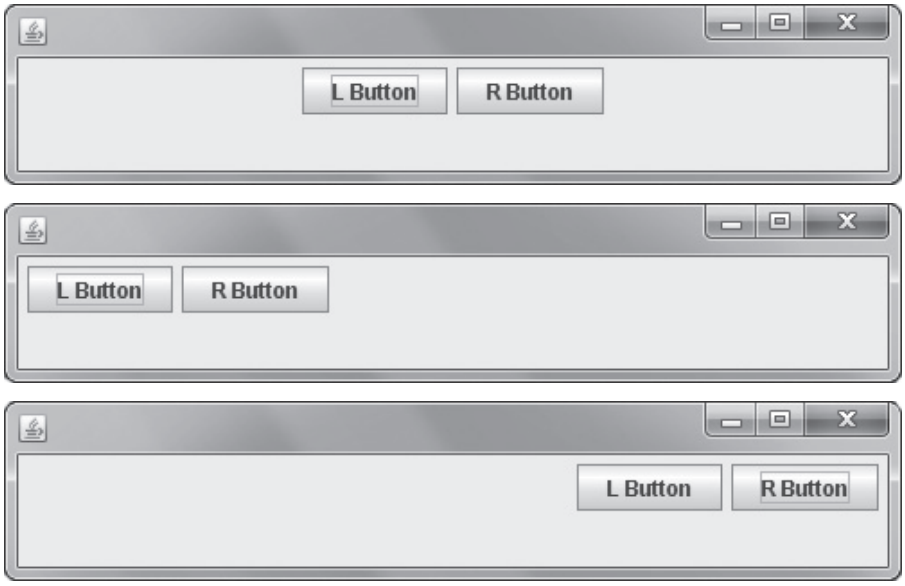

Figure 15-9 The `JDemoFlowLayout` application as it first appears on the screen, after the user chooses the "L" Button, and after the user chooses the "R" Button

Using `GridLayout`

If you want to arrange components into equal rows and columns, you can use the **`GridLayout` manager** 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 following statement establishes an anonymous `GridLayout` with four horizontal rows and five vertical columns in a `Container` named `con`:

```
con.setLayout(new GridLayout(4, 5));
```

Specifying rows and then columns when you use `GridLayout` might seem natural to you, because this is the approach you take when defining two-dimensional arrays.

As you add new `Components` to a `GridLayout`, they are positioned in sequence from left to right across each row. Unfortunately, you can't skip a position or specify an exact position for a component. (However, you can add a blank label to a grid position to give the illusion of skipping a position.) You also can specify a vertical and horizontal gap measured in pixels, using two additional arguments. For example, Figure 15-10 shows a `JDemoGridLayout` program that uses the shaded statement to establish a `GridLayout` with three horizontal rows and two vertical columns, and horizontal and vertical gaps of five pixels each. Five `JButton Components` are added to the `JFrame`'s automatically retrieved content pane.

```
import javax.swing.*;
import java.awt.*;
public class JDemoGridLayout extends JFrame
{
   private JButton b1 = new JButton("Button 1");
   private JButton b2 = new JButton("Button 2");
   private JButton b3 = new JButton("Button 3");
   private JButton b4 = new JButton("Button 4");
   private JButton b5 = new JButton("Button 5");
   private GridLayout layout = new GridLayout(3, 2, 5, 5);
   private Container con = getContentPane();
   public JDemoGridLayout()
   {
      con.setLayout(layout);
      con.add(b1);
      con.add(b2);
      con.add(b3);
      con.add(b4);
      con.add(b5);
      setSize(200, 200);
   }
   public static void main(String[] args)
   {
      JDemoGridLayout frame = new JDemoGridLayout();
      frame.setVisible(true);
   }
}
```

Figure 15-10 The `JDemoGridLayout` class

Figure 15-11 shows the output of the `JDemoGridLayout` application. The `Components` are placed into the pane across the three rows. Because there are six positions but only five `Components`, one spot remains unused.

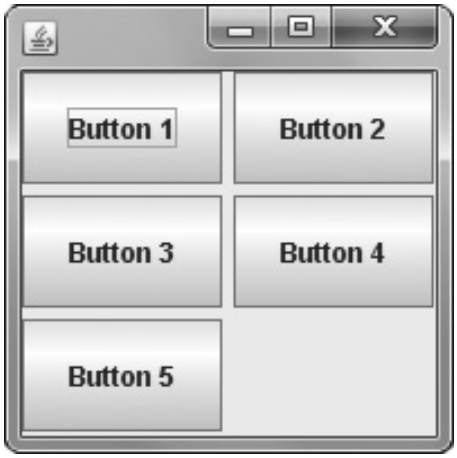

Figure 15-11 Output of the `JDemoGridLayout` program

With `GridLayout`, you can specify the number of rows and use 0 for the number of columns to let the layout manager determine the number of columns, or you can use 0 for the number of rows, specify the number of columns, and let the layout manager calculate the number of rows.

When trying to decide whether to use `GridLayout` or `FlowLayout`, remember the following:

- Use `GridLayout` when you want components in fixed rows and columns and you want the components' size to fill the available space.
- Use `FlowLayout` if you want Java to determine the rows and columns, do not want a rigid row and column layout, and want components to retain their "natural" size so their contents are fully visible.

Using `CardLayout`

The **`CardLayout` manager** generates a stack of containers or components, one on top of another, much like a blackjack dealer reveals playing cards one at a time from the top of a deck. Each component in the group is referred to as a card, and each card can be any component type—for example, a `JButton`, `JLabel`, or `JPanel`. You use a `CardLayout` when you want multiple components to share the same display space.

A card layout is created from the `CardLayout` class using one of two constructors:

- `CardLayout()` creates a card layout without a horizontal or vertical gap.
- `CardLayout(int hgap, int vgap)` creates a 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, Figure 15-12 shows a `JDemoCardLayout` class that uses a `CardLayout` manager to create a stack of `JButtons` that contain the labels "Ace of Hearts", "Three of Spades", and "Queen of Clubs". In the class constructor, you need a slightly different version of the `add()` method to add a component to a content pane whose layout manager is `CardLayout`. The format of the method is:

```
add(aString, aContainer);
```

In this statement, `aString` represents a name you want to use to identify the `Component` card that is added.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoCardLayout extends JFrame
   implements ActionListener
{
   private CardLayout cards = new CardLayout();
   private JButton b1 = new JButton("Ace of Hearts");
   private JButton b2 = new JButton("Three of Spades");
   private JButton b3 = new JButton("Queen of Clubs");
   private Container con = getContentPane();
   public JDemoCardLayout()
   {
      con.setLayout(cards);
      con.add("ace", b1);
      b1.addActionListener(this);
      con.add("three", b2);
      b2.addActionListener(this);
      con.add("queen", b3);
      b3.addActionListener(this);
      setSize(200, 100);
   }
   public void actionPerformed(ActionEvent e)
   {
      cards.next(getContentPane());
   }
   public static void main(String[] args)
   {
      JDemoCardLayout frame = new JDemoCardLayout();
      frame.setVisible(true);
   }
}
```

Figure 15-12 The `JDemoCardLayout` class

In a program that has a `CardLayout` manager, a change of cards is usually triggered by a user's action. For example, in the `JDemoCardLayout` program, each `JButton` can trigger the `actionPerformed()` method. Within this method, the statement `next(getContentPane())` flips to the next card of the container. (The order of the cards depends on the order in which you add them to the container.) You also can use `previous(getContentPane());`, `first(getContentPane());`, and `last(getContentPane());` to flip to the previous, first, and last card, respectively. You can go to a specific card by using the `String` name assigned in the `add()` method call. For example, in the application in Figure 15-12, the following statement would display "Three of Spades" because `"three"` is used as the first argument when the `b2` object is added to the content pane in the `JDemoCardLayout` constructor:

```
cards.show(getContentPane(), "three");
```

Figure 15-13 shows the output of the `JDemoCardLayout` program when it first appears on the screen, after the user clicks the button once, and after the user clicks the button a second time. Because each `JButton` is a card, each `JButton` consumes the entire viewing area in the container that uses the `CardLayout` manager. If the user continued to click the card buttons in Figure 15-13, the cards would continue to cycle in order.

Figure 15-13 Output of `JDemoCardLayout` when it first appears on the screen, after the user clicks once, and after the user clicks twice

The `JTabbedPane` class operates like a container with a `CardLayout`, but folder-type tabs are in place for the user to select the various components. You can find out more about the class at www.oracle.com/technetwork/java/index.html.

Using Advanced Layout Managers

Just as professional Java programmers are constantly creating new `Components`, they also create new layout managers. You are certain to encounter new and interesting layout managers during your programming career; you might even create your own.

For example, when `GridLayout` is not sophisticated enough for your purposes, you can use `GridBagLayout`. The **`GridBagLayout` manager** 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, and more than 20 methods are associated with the class. Visit the Java Web site for details on how to use this class.

Another layout manager option is the **`BoxLayout` manager**, which allows multiple components to be laid out either vertically or horizontally. The components do not wrap, so a vertical arrangement of components, for example, stays vertically arranged when the frame is resized. The Java Web site can provide you with details.

Watch the video *Layout Managers*.

TWO TRUTHS & A LIE

Learning More About Layout Managers

1. The `FlowLayout` manager is the default manager class for all content panes.
2. The `BorderLayout` manager can directly hold only up to five components.
3. The `GridLayout` manager arranges components in rows and columns.

The false statement is #1. The `BorderLayout` manager is the default manager class for all content panes.

You Do It

Using `BorderLayout`

Using layout managers in the containers in your applications allows flexibility in arranging the components that users see on the screen. In this section, you create a `JFrame` that uses a `BorderLayout` and place components in each region. In the following sections, you will observe how the same components appear when other layout managers are used.

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

```
import javax.swing.*;
import java.awt.*;
public class JBorderLayout extends JFrame
{
```

(continues)

(continued)

2. Instantiate five `JButton` objects, each with a label that is the name of one of the regions used by `BorderLayout`:

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

3. Write the constructor that sets the `JFrame`'s layout manager and adds each of the five `JButton`s to the appropriate region. Also set the default close operation for the `JFrame`.

```
public JBorderLayout()
{
   setLayout(new BorderLayout());
   add(nb, BorderLayout.NORTH);
   add(sb, BorderLayout.SOUTH);
   add(eb, BorderLayout.EAST);
   add(wb, BorderLayout.WEST);
   add(cb, BorderLayout.CENTER);
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

4. Add a `main()` method that instantiates a `JBorderLayout` object and sets its size and visibility, and include a closing curly brace for the class:

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

5. Save the file as **JBorderLayout.java**, and then compile and execute it. The output looks like Figure 15-14. Each `JButton` entirely fills its region. (If you click the `JButton`s, they appear to be pressed, but because you have not implemented `ActionListener`, no other action is taken.)

(continues)

(continued)

Figure 15-14 Output of the `JBorderLayout` program

6. So you can observe the effects of changing the size of the viewing area, use your mouse to drag the right border of the `JFrame` to increase the width to approximately that shown in Figure 15-15. Notice that the center region expands, while the east and west regions retain their original size.

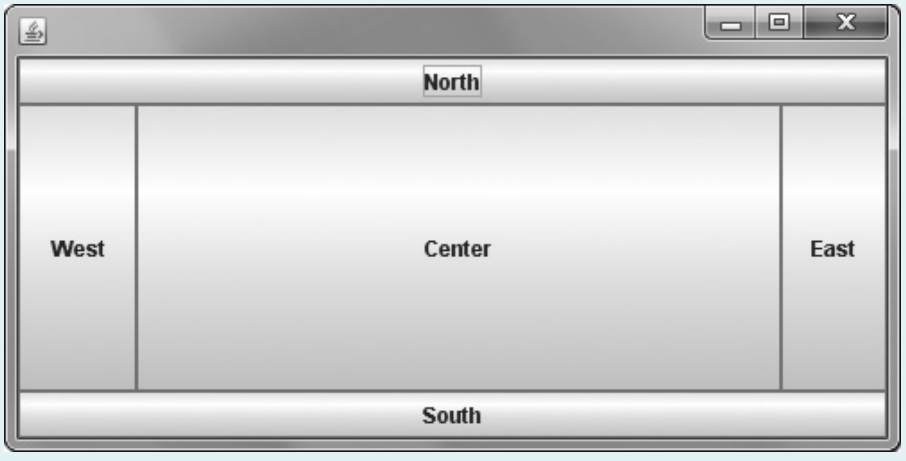

Figure 15-15 Output of the `JBorderLayout` program after the user drags the right border to increase the width

(continues)

(continued)

7. Experiment with resizing both the width and height of the `JFrame`. Close the `JFrame` when you finish.

Using Fewer than Five Components with the `BorderLayout` Manager

When you use `JBorderLayout`, you are not required to place components in every region. For example, you might use only four components, leaving the north region empty. Next, you remove one of the objects from the `JBorderLayout JFrame` to observe the effect.

1. Open the **JBorderLayout.java** file in your text editor. Immediately save it as **JBorderLayoutNoNorth.java**.
2. Change the class name to **`JBorderLayoutNoNorth`**. Also change the constructor name and the two instances of the class name in the `main()` method.
3. Remove the declaration of the "North" button, and within the constructor, remove the statement that adds the "North" button to the `JFrame`.
4. Save the file, compile it, and then run the program. The output appears as shown in Figure 15-16. The center region occupies the space formerly held by the north region.

Figure 15-16 Output of the `JBorderLayoutNoNorth` program

5. Experiment with removing some of the other components from the `JBorderLayoutNoNorth` program.

(continues)

(continued)

Using `FlowLayout`

Next, you modify the `JBorderLayout` program to demonstrate how the same components appear when using `FlowLayout`.

1. Open the **JBorderLayout.java** file in your text editor, and immediately save it as **JFlowLayoutRight.java**.
2. Change the class name from `JBorderLayout` to **`JFlowLayoutRight`**. Also change the constructor name and the references to the name in the `main()` method.
3. Within the constructor, change the `setLayout()` statement to use `FlowLayout` and right alignment:

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

4. Alter each of the five `add()` statements so that just the button name appears within the parentheses and the region is omitted. For example, `add(nb, BorderLayout.NORTH);` becomes the following:

```
add(nb);
```

5. Save the file, and then compile and execute it. Your output should look like Figure 15-17. The components have their "natural" size (or preferred size)—the minimum size the buttons need to display their labels. The buttons flow across the `JFrame` surface in a row until no more can fit; in Figure 15-17 the last two buttons added cannot fit in the first row, so they appear in the second row, right-aligned.

Figure 15-17 Output of the `JFlowLayoutRight` program

(continues)

(continued)

6. Experiment with widening and narrowing the `JFrame`, and observe how the components realign. Then close the `JFrame`.

Using `GridLayout`

Next, you modify a `JFrame` to demonstrate `GridLayout`.

1. Open the **JFlowLayoutRight.java** file in your text editor, and save the file as **JGridLayout.java**.
2. Change the class name from `JFlowLayoutRight` to **`JGridLayout`**. Change the constructor name and the two references to the class in the `main()` method.
3. Within the constructor, change the `setLayout()` statement to establish a `GridLayout` with two rows, three columns, a horizontal space of two pixels, and a vertical space of four pixels:

```
setLayout(new GridLayout(2, 3, 2, 4));
```

4. Save the file, and then compile and execute it. The components are arranged in two rows and three columns from left to right across each row, in the order they were added to their container. Because there are only five components, one grid position still is available. See Figure 15-18.

Figure 15-18 Output of the `JGridLayout` program

5. Close the program.

(continues)

(continued)

Using CardLayout

Next, you create a `CardLayout` with five cards, each holding one of the `JButton`s used in the previous examples.

1. Open the **JGridLayout.java** file in your text editor, and save the file as **JCardLayout.java**.
2. Change the class name from `JGridLayout` to **`JCardLayout`**. Also change the constructor name and the two references in the `main()` method.
3. Within the constructor, change the `setLayout()` statement to establish a `CardLayout`:

```
setLayout(new CardLayout());
```

4. Change the five `add()` statements that add the buttons to the content pane so that each includes a `String` that names the added component, as follows:

```
add("north", nb);
add("south", sb);
add("east", eb);
add("west", wb);
add("center", cb);
```

5. Save the file, and then compile and execute it. The output looks like Figure 15-19. You see only the "North" `JButton` because, as the first one added, it is the top card. You can click the button, but no actions take place because you have not implemented `ActionListener`.

Figure 15-19 Output of the `JCardLayout` program

(continues)

(continued)

6. Close the program.

Viewing All the Cards in `CardLayout`

Next, you modify the `JCardLayout` program so that its buttons can initiate events that allow you to view all five `JButton`s you add to the content pane.

1. Open the **JCardLayout.java** file in your text editor, and save the file as **JCardLayout2.java**.
2. Change the class name, constructor name, and two `main()` method references from `JCardLayout` to **`JCardLayout2`**.
3. At the top of the file, add the `import` statement that adds the classes and methods that allow the class to respond to events:

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

4. At the end of the class header, insert the following phrase so the `JFrame` can respond to button clicks:

```
implements ActionListener
```

5. Instead of an anonymous layout manager, you need to create a `CardLayout` manager with an identifier that you can use with the `next()` method when the user clicks a button. Immediately after the five `JButton` declaration statements, insert the following statement:

```
CardLayout cardLayout = new CardLayout();
```

6. Within the constructor, change the `setLayout()` statement so it uses the named layout manager:

```
setLayout(cardLayout);
```

7. At the end of the constructor, add five statements that allow each of the buttons to initiate an `ActionEvent`:

```
nb.addActionListener(this);
sb.addActionListener(this);
eb.addActionListener(this);
wb.addActionListener(this);
cb.addActionListener(this);
```

(continues)

(continued)

8. After the constructor's closing curly brace, add an `actionPerformed()` method that responds to user clicks. The method uses the `next()` method to display the next card (next button) in the collection.

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

9. Save, compile, and run the program. The output looks the same as in Figure 15-19: you see only the "North" `JButton`. However, when you click it, the button changes to "South", "East", "West", and "Center" in succession. Close the `JFrame` when you finish.

Using the `JPanel` Class

Using the `BorderLayout`, `FlowLayout`, `GridLayout`, and `CardLayout` managers would provide a limited number of screen arrangements if you could place only one `Component` in a section of the layout. Fortunately, you can greatly increase the number of possible component arrangements by using the `JPanel` class. A **`JPanel`** is a plain, borderless surface that can hold lightweight UI components. Figure 15-20 shows the inheritance hierarchy of the `JPanel` class. You can see that every `JPanel` is a `Container`; you use a `JPanel` to hold other UI components, such as `JButtons`, `JCheckBoxes`, or even other `JPanels`. By using `JPanels` within `JPanels`, you can create an infinite variety of screen layouts. The default layout manager for every `JPanel` is `FlowLayout`.

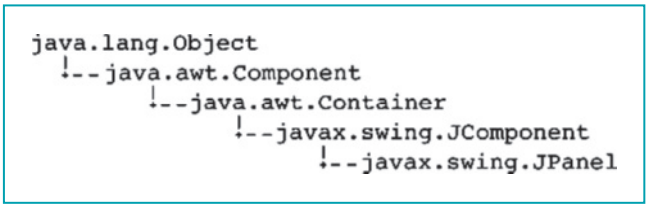

Figure 15-20 The inheritance hierarchy of the `JPanel` class

To add a component to a `JPanel`, you call the container's `add()` method, using the component as the argument. For example, Figure 15-21 shows the code that creates a `JFrameWithPanels` class that extends `JFrame`. A `JButton` is added to a `JPanel` named `panel1`, and two more `JButtons` are added to another `JPanel` named `panel2`. Then `panel1` and `panel2` are added to the `JFrame`'s content pane.

```
import javax.swing.*;
import java.awt.*;
import java.awt.Color;
public class JFrameWithPanels extends JFrame
{
   private final int WIDTH = 250;
   private final int HEIGHT = 120;
   private JButton button1 = new JButton("One");
   private JButton button2 = new JButton("Two");
   private JButton button3 = new JButton("Three");
   public JFrameWithPanels()
   {
      super("JFrame with Panels");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JPanel panel1 = new JPanel();
      JPanel pane12 = new JPanel();
      Container con = getContentPane();
      con.setLayout(new FlowLayout());
      con.add(panel1);
      con.add(panel2);
      panel1.add(button1);
      panel1.setBackground(Color.BLUE);
      panel2.add(button2);
      panel2.add(button3);
      panel2.setBackground(Color.BLUE);
      setSize(WIDTH, HEIGHT);
   }
   public static void main(String[] args)
   {
      JFrameWithPanels frame = new JFrameWithPanels();
      frame.setVisible(true);
   }
}
```

Figure 15-21 The `JFrameWithPanels` class

Figure 15-22 shows the output of the `JFrameWithPanels` program. Two `JPanels` have been added to the `JFrame`. Because this application uses the `setBackground()` method to make each `JPanel`'s background blue, you can see where one panel ends and the other begins. The first `JPanel` contains a single `JButton`, and the second one contains two `JButtons`.

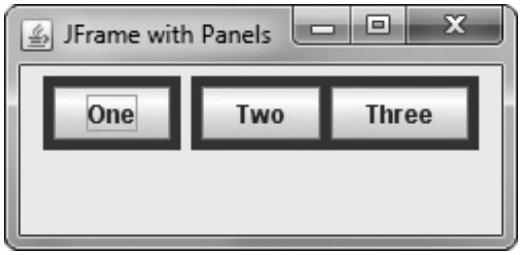

Figure 15-22 Output of the `JFrameWithPanels` application

When you create a `JPanel` object, you can use one of four constructors. The different constructors allow you to use default values or to specify a layout manager and whether the `JPanel` is double buffered. If you indicate **double buffering**, which is the default buffering strategy, you specify that additional memory space will be used to draw the `JPanel` offscreen when it is updated. With double buffering, a redrawn `JPanel` is displayed only when it is complete; this provides the viewer with updated screens that do not flicker while being redrawn. The four constructors are as follows:

- `JPanel()` creates a `JPanel` with double buffering and a flow layout.
- `JPanel(LayoutManager layout)` creates a `JPanel` with the specified layout manager and double buffering.
- `JPanel(Boolean isDoubleBuffered)` creates a `JPanel` with a flow layout and the specified double-buffering strategy.
- `JPanel(LayoutManager layout, Boolean isDoubleBuffered)` creates a `JPanel` with the specified layout manager and the specified buffering strategy.

When you employ double buffering, the visible screen surface is called the **primary surface**, and the offscreen image is called the **back buffer**. The act of copying the contents from one surface to another is frequently referred to as a **block line transfer**, or **blitting**, because of the acronym *blt*, pronounced *blit*. Double buffering prevents "tearing," the visual effect that occurs when you see parts of different images because the redrawing rate is not fast enough. As with most beneficial features, double buffering has a cost: additional memory requirements.

As with many aspects of Java, there are multiple ways to achieve the same results. For example, each of the following techniques creates a `JPanel` that uses a `BorderLayout` manager:

- You can create a named layout and use it as an argument in a `JPanel` constructor:

```
BorderLayout border = new BorderLayout();
JPanel myPanel = new JPanel(border);
```

- You can use an anonymous layout manager in the `JPanel` constructor:

```
JPanel myPanel = new JPanel(new BorderLayout());
```

- You can create a JPanel and then set its layout manager using the setLayout() method:

```
JPanel myPanel = new JPanel();
myPanel.setLayout(new BorderLayout());
```

When a JPanel will have a layout other than FlowLayout, specifying the layout manager when you create the JPanel is preferable for performance reasons. If you create the JPanel first and change its layout later, you automatically create an unnecessary FlowLayout object for the original instantiation.

You add components to a JPanel with the add() method. Figure 15-23 shows a JDemoManyPanels program in which the JFrame contains four JPanels and 12 JButtons that each display a single spelled-out number so you can better understand their positions. The automatically supplied content pane for the JFrame is assigned a BorderLayout, and each JPanel is assigned either a GridLayout or FlowLayout and placed in one of the regions (leaving the north region empty). One or more JButtons are then placed on each JPanel. Figure 15-24 shows the output as the user adjusts the borders of the JFrame to change its size. Using the code as a guide, be certain you understand why each JButton appears as it does in the JFrame.

```
import javax.swing.*;
import java.awt.*;
public class JDemoManyPanels extends JFrame
{
 // Twelve buttons
   private JButton button01 = new JButton("One");
   private JButton button02 = new JButton("Two");
   private JButton button03 = new JButton("Three");
   private JButton button04 = new JButton("Four");
   private JButton button05 = new JButton("Five");
   private JButton button06 = new JButton("Six");
   private JButton button07 = new JButton("Seven");
   private JButton button08 = new JButton("Eight");
   private JButton button09 = new JButton("Nine");
   private JButton button10 = new JButton("Ten");
   private JButton button11 = new JButton("Eleven");
   private JButton button12 = new JButton("Twelve");

 // Four panels
   private JPanel panel01 = new JPanel(new GridLayout(2, 0));
   private JPanel panel02 = new JPanel(new FlowLayout());
   private JPanel panel03 = new JPanel(new FlowLayout());
   private JPanel panel04 = new JPanel(new GridLayout(2, 0));
```

Figure 15-23 The JDemoManyPanels class *(continues)*

(*continued*)

```
    public JDemoManyPanels()
    {
       setLayout(new BorderLayout());
       add(panel01, BorderLayout.WEST);
       add(panel02, BorderLayout.CENTER);
       add(panel03, BorderLayout.SOUTH);
       add(panel04, BorderLayout.EAST);

       panel01.add(button01);
       panel01.add(button02);
       panel01.add(button03);

       panel02.add(button04);
       panel02.add(button05);
       panel02.add(button06);

       panel03.add(button07);

       panel04.add(button08);
       panel04.add(button09);
       panel04.add(button10);
       panel04.add(button11);
       panel04.add(button12);

       setSize(400, 250);
    }
    public static void main(String[] args)
    {
       JDemoManyPanels frame = new JDemoManyPanels();
       frame.setVisible(true);
    }
}
```

Figure 15-23 The `JDemoManyPanels` class

If you were creating a program with as many buttons and panels as the one in Figure 15-23, you might prefer to create arrays of the components instead of so many individually named ones. This example does not use an array so you can more easily see how each component is placed.

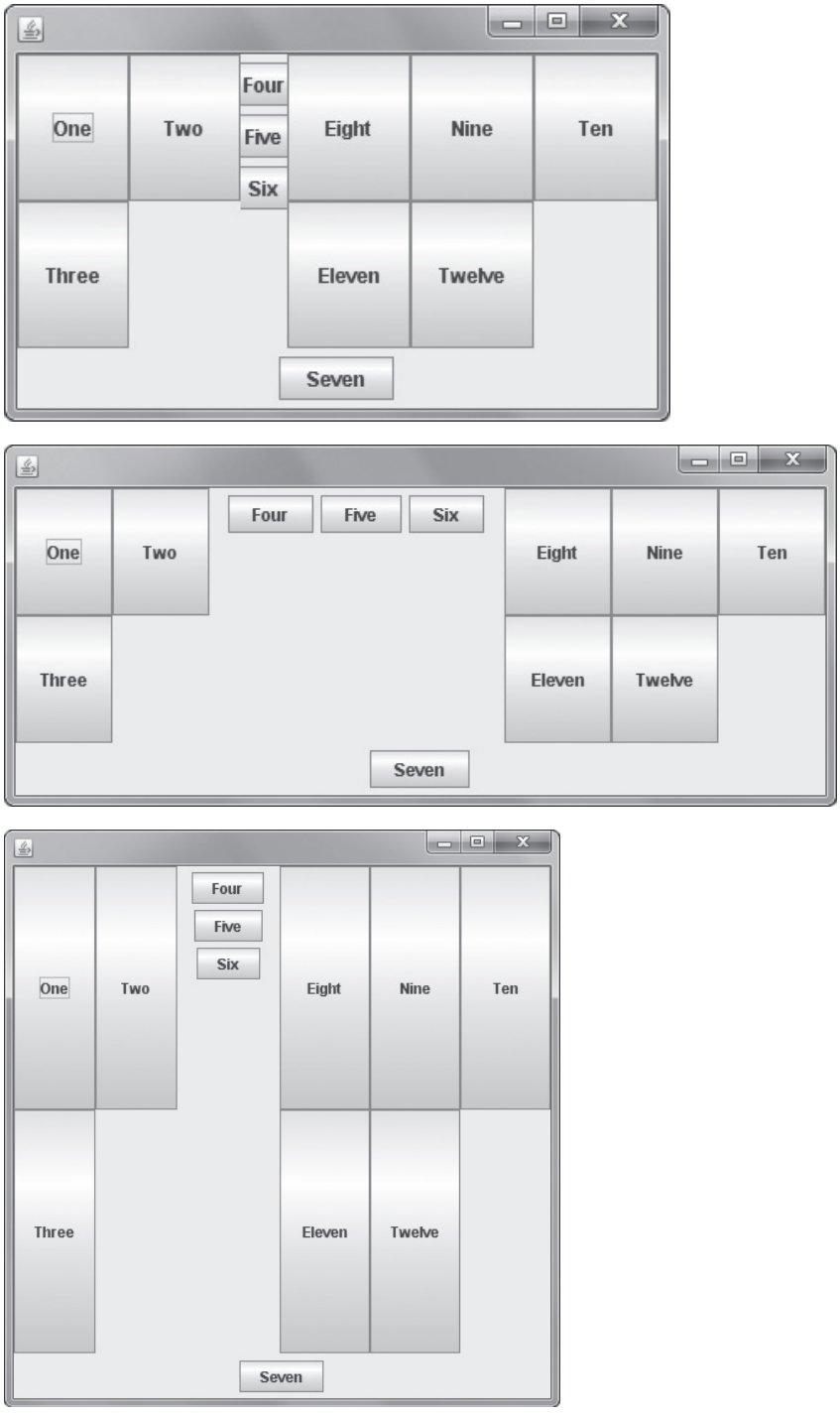

Figure 15-24 Output of the `JDemoManyPanels` program: three views as the user adjusts the `JFrame` borders

Swing containers other than JPanel and content panes generally provide Application Program Interface (API) methods that you should use instead of the add() method. See the Java Web site for details.

GridLayout provides you with rows and columns that are similar to a two-dimensional array. Therefore, it particularly lends itself to displaying arrays of objects. For example, Figure 15-25 contains a Checkerboard class that displays a pattern of eight rows and columns in alternating colors. The JPanel placed in the content pane has a GridLayout of eight by eight. Sixty-four JPanels are declared, and in a loop, one by one, they are instantiated and assigned to a section of the grid (see shaded statements). After each set of eight JPanels is assigned to the grid (when x is evenly divisible by 8), the first and second color values are reversed, so that the first row starts with a blue square, the second row starts with a white square, and so on. Within each row, all the even-positioned squares are filled with one color, and the odd-positioned squares are filled with the other. Figure 15-26 shows the output.

```
import java.awt.*;
import javax.swing.*;
import java.awt.Color;
public class Checkerboard extends JFrame
{
   private final int ROWS = 8;
   private final int COLS = 8;
   private final int GAP = 2;
   private final int NUM = ROWS * COLS;
   private int x;
   private JPanel pane = new JPanel
      (new GridLayout(ROWS, COLS, GAP, GAP));
   private JPanel[] panel = new JPanel[NUM];
   private Color color1 = Color.WHITE;
   private Color color2 = Color.BLUE;
   private Color tempColor;
   public Checkerboard()
   {
      super("Checkerboard");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      add(pane);
      for(x = 0; x < NUM; ++x)
      {
         panel[x] = new JPanel();
         pane.add(panel[x]);
         if(x % COLS == 0)
         {
            tempColor = color1;
            color1 = color2;
            color2 = tempColor;
         }
```

Figure 15-25 The Checkerboard class *(continues)*

(*continued*)

```
            if(x % 2 == 0)
               panel[x].setBackground(color1);
            else
               panel[x].setBackground(color2);
         }
      }
      public static void main(String[] args)
      {
         Checkerboard frame = new Checkerboard();
         final int SIZE = 300;
         frame.setSize(SIZE, SIZE);
         frame.setVisible(true);
      }
   }
```

Figure 15-25 The Checkerboard class

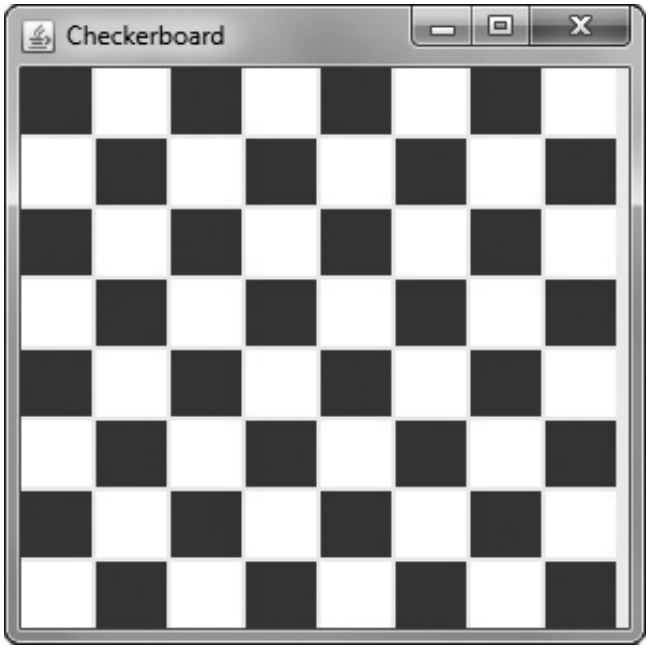

Figure 15-26 Output of the Checkerboard application

When creating the `Checkerboard` class, you might be tempted to create just two `JPanel`s, one blue and one white, and add them to the content pane multiple times. However, each GUI component can be contained only once. If a component is already in a container and you try to add it to another container, the component will be removed from the first container and then added to the second.

Watch the video *The `JPanel` Class*.

TWO TRUTHS & A LIE

Using the `JPanel` Class

1. A `JPanel` is a plain, borderless surface that can hold lightweight UI components.
2. To add a component to a `JPanel`, you call the component's `add()` method, using the `JPanel` as the argument.
3. Different `JPanel` constructors allow you to use default values or to specify a layout manager and whether the `JPanel` is double buffered.

The false statement is #2. To add a component to a `JPanel`, you call the container's `add()` method, using the component as the argument.

Creating `JScrollPanes`

When components in a `Swing` UI require more display area than they have been allocated, you can use a `JScrollPane` container to hold the components in a way that allows a user to scroll initially invisible parts of the pane into view. A **`JScrollPane`** provides scroll bars along the side or bottom of a pane, or both, with a viewable area called a **viewport**. Figure 15-27 displays the inheritance hierarchy of the `JScrollPane` class.

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

Figure 15-27 The inheritance hierarchy of the `JScrollPane` class

The JScrollPane constructor takes one of four forms:

- JScrollPane() creates an empty JScrollPane in which both horizontal and vertical scroll bars appear when needed.
- JScrollPane(Component) creates a JScrollPane that displays the contents of the specified component.
- JScrollPane(Component, int, int) creates a JScrollPane that displays the specified component and includes both vertical and horizontal scroll bar specifications.
- JScrollPane(int, int) creates a JScrollPane with both vertical and horizontal scroll bar specifications.

When you create a simple scroll pane using the constructor that takes no arguments, as in the following example, horizontal and vertical scroll bars appear only if they are needed, that is, if the contents of the pane cannot be fully displayed without them:

```
JScrollPane aScrollPane = new JScrollPane();
```

To force the display of a scroll bar, you can use class variables defined in the ScrollPaneConstants class, as follows:

```
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED
ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS
ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER
```

For example, the following code creates a scroll pane that displays an image named picture, a vertical scroll bar, and no horizontal scroll bar:

```
JScrollPane scroll = new JScrollPane(picture,
   ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
   ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

Figure 15-28 shows a JScrollDemo class in which a label with a large font is added to a panel. The scroll pane named scroll includes the panel and two scroll bars.

```
import javax.swing.*;
import java.awt.*;
public class JScrollDemo extends JFrame
{
   private JPanel panel = new JPanel();
   private JScrollPane scroll = new JScrollPane(panel,
      ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
      ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
```

Figure 15-28 The JScrollDemo application *(continues)*

(*continued*)


```
    private JLabel label = new JLabel("Four score and seven");
    private Font bigFont = new Font("Arial", Font.PLAIN, 20);
    private Container con;
    public JScrollDemo()
    {
       super("JScrollDemo");
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       con = getContentPane();
       label.setFont(bigFont);
       con.add(scroll);
       panel.add(label);
    }
    public static void main(String[] args)
    {
       final int WIDTH = 180;
       final int HEIGHT = 100;
       JScrollDemo aFrame = new JScrollDemo();
       aFrame.setSize(WIDTH, HEIGHT);
       aFrame.setVisible(true);
    }
}
```

Figure 15-28 The `JScrollDemo` application

The `JScrollDemo` object in the program in Figure 15-28 is purposely set small enough (180 × 100) so that only part of the label it contains is visible at a time. A user can slide the scroll bars to view the entire label. Figure 15-29 shows the output with the scroll bar in two positions.

Figure 15-29 Output of the `JScrollDemo` application

TWO TRUTHS & A LIE

Creating JScrollPanes

1. A JScrollPane can provide scroll bars along the side or bottom of a pane, or both.
2. When you create a simple scroll pane using the constructor that takes no arguments, horizontal and vertical scroll bars appear only if they are needed.
3. You cannot force the display of a scroll bar in a JScrollPane unless the components it contains require too much room.

The false statement is #3. You can use class variables defined in the ScrollPaneConstants class to force the display of a scroll bar.

A Closer Look at Events and Event Handling

In the chapter *Introduction to Swing Components*, you worked with ActionEvents and ItemEvents that are generated when a user works with a control that is included in one of your programs. The parent class for all events is EventObject, which descends from the Object class. EventObject is the parent of AWTEvent, which in turn is the parent of specific event classes such as ActionEvent and ItemEvent. The abstract class AWTEvent is contained in the package java.awt.event. Although you might think it would have been logical for the developers to name the event base class Event, there is no currently active, built-in Java class named Event (although there was one in Java 1.0). Figure 15-30 illustrates the inheritance hierarchy 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 15-30 The inheritance hierarchy of event classes

You can see in Figure 15-30 that `ComponentEvent` is a parent to several event classes, including `InputEvent`, which is a parent of `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` are generated by components that users can click, such as `JButtons` and `JCheckBoxes`, and `TextEvents` are generated by components into which the user enters text, such as a `JTextField`. `MouseEvents` include determining the location of the mouse pointer and distinguishing between a single- and double-click. Table 15-3 lists some common user actions and the events that are generated from them.

User Action	Resulting Event Type
Click a button	`ActionEvent`
Click a component	`MouseEvent`
Click an item in a list box	`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`

Table 15-3 Examples of user actions and their resulting event types

Because `ActionEvents` involve the mouse, it is easy to confuse `ActionEvents` and `MouseEvents`. If you are interested in `ActionEvents`, you focus on changes in a component (for example, a `JButton` on a `JFrame` being pressed); if you are interested in `MouseEvents`, your focus is on what the user does manually with the mouse (for example, clicking the left mouse button).

When you write programs with GUIs, 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 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 that produces no effect when clicked, you simply don't look for a message to occur.

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 15-3 has a listener interface associated with it, so that for every event class, `<name>Event`, there is a similarly named `<name>Listener` interface. For example, `ActionEvent` has an `ActionListener` interface. (The `MouseEvent` class has an additional listener besides `MouseListener`: `MouseMotionListener`.)

Remember that an interface contains only abstract methods, so 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 can leave the methods empty in your implementation, providing a header and curly braces but no statements.

Every `<name>Listener` interface method has the 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 an event handler 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. Instead of implementing a listener class, you can extend an adapter class. An **adapter class** implements all the methods in an interface, providing an empty body for each method. For example, the `MouseAdapter` class provides an empty method for all the methods contained in `MouseListener`. The advantage to extending an adapter class instead of implementing a listener class is that you need to write only the methods you want to use, and you do not have to bother creating empty methods for all the others. (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 `JFrame` is an `ActionListener` and contains a `JButton` named `pushMe`, then the following statement registers this `JFrame` as a listener for the `pushMe JButton`:

```
pushMe.addActionListener(this);
```

Table 15-4 lists the events with their listeners and handlers.

Event	Listener(s)	Handler(s)
ActionEvent	ActionListener	actionPerformed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
TextEvent	TextListener	textValueChanged(TextEvent)
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
ContainerEvent	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
ComponentEvent	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
FocusEvent	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
MouseEvent	MouseListener MouseMotionListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent) mouseDragged(MouseEvent) mouseMoved(MouseEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent) keyTyped(KeyEvent) keyReleased(KeyEvent)
WindowEvent	WindowListener	windowActivated(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)
MouseWheelEvent	MouseWheelListener	mouseWheelMoved(MouseWheelEvent)

Table 15-4 Events with their related listeners and handlers

An Event-Handling Example: 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 press 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 which character was typed. When the user presses a key that does not generate a character, such as a function key (sometimes called an **action key**), `keyTyped()` does not execute. The methods `keyPressed()` and `keyReleased()` provide the only ways to get information about keys that don't generate characters. The `KeyEvent` class contains constants known as **virtual key codes** that represent keyboard keys that have been pressed. For example, when you type *A*, two virtual key codes are generated: Shift and "a". The virtual key code constants have names such as `VK_SHIFT` and `VK_ALT`. See the Java Web site for a complete list of virtual key codes. Figure 15-31 shows a `JDemoKeyFrame` class that uses the `keyTyped()` method to discover which key the user typed last.

Java programmers call `keyTyped()` events "higher-level" events because they do not depend on the platform or keyboard layout. (For example, the key that generates `VK_Q` on a U.S. keyboard layout generates `VK_A` on a French keyboard layout.) In contrast, `keyPressed()` and `keyReleased()` events are "lower-level" events and do depend on the platform and keyboard layout. According to the Java documentation, using `keyTyped()` is the preferred way to find out about character input.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoKeyFrame extends JFrame
   implements KeyListener
{
   private JLabel prompt = new JLabel("Type keys below:");
   private JLabel outputLabel = new JLabel();
   private JTextField textField = new JTextField(10);
   public JDemoKeyFrame()
   {
      setTitle("JKey Frame");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new BorderLayout());
      add(prompt, BorderLayout.NORTH);
      add(textField, BorderLayout.CENTER);
      add(outputLabel, BorderLayout.SOUTH);
      addKeyListener(this);
      textField.addKeyListener(this);
   }
   public void keyTyped(KeyEvent e)
   {
      char c = e.getKeyChar();
      outputLabel.setText("Last key typed: " + c);
   }
```

Figure 15-31 The `JDemoKeyFrame` class *(continues)*

(*continued*)

```
    public void keyPressed(KeyEvent e)
    {
    }
    public void keyReleased(KeyEvent e)
    {
    }
    public static void main(String[] args)
    {
       JDemoKeyFrame keyFrame = new JDemoKeyFrame();
       final int WIDTH = 250;
       final int HEIGHT = 100;
       keyFrame.setSize(WIDTH, HEIGHT);
       keyFrame.setVisible(true);
    }
}
```

Figure 15-31 The `JDemoKeyFrame` class

A prompt in the north border area asks the user to type in the text field in the center area. With each key press by the user, the `keyTyped()` method changes the label in the south border area of the frame to display the key that generated the most recent `KeyEvent`. Figure 15-32 shows the output after the user has typed several characters into the text field.

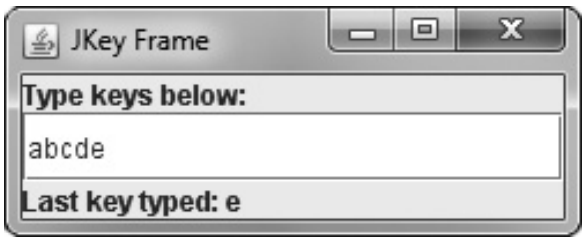

Figure 15-32 Output of the `JDemoKeyFrame` application after the user has typed several characters

Watch the video *Event Handling*.

TWO TRUTHS & A LIE

A Closer Look at Events and Event Handling

1. ActionEvents are generated by components that users can click, TextEvents are generated by components into which the user enters text, and MouseEvents are generated by mouse actions.
2. Every <name>Listener interface method has a return type that refers to an instance of the corresponding <name>Event class.
3. An adapter class implements all the methods in an interface, providing an empty body for each method.

The false statement is #2. Every <name>Listener interface method has the return type void, and each takes one argument: an object that is an instance of the corresponding <name>Event class.

Using AWTEvent Class Methods

In addition to the handler methods included with the event listener interfaces, the AWTEvent classes themselves contain many other methods that return information about an event. For example, the ComponentEvent class contains a getComponent() method that allows you to determine which of multiple Components generates an event. The WindowEvent class contains a similar method, getWindow(), that returns the Window that is the source of an event. Table 15-5 lists some useful methods for many of the event classes. All Components have these methods:

- addComponentListener()
- addFocusListener()
- addMouseListener()
- addMouseMotionListener()

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 the Alt key was pressed when the event occurred
InputEvent	boolean isControlDown()	Returns whether the Ctrl key was pressed when the event occurred
InputEvent	boolean isShiftDown()	Returns whether the Shift key was pressed 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 the x- and y-coordinates of the mouse location

Table 15-5 Useful event class methods

You can call any of the methods listed in Table 15-5 by using the object-dot-method format that you use with all class methods. For example, if you have a KeyEvent named inputEvent and an integer named unicodeVal, the following statement is valid:

```
unicodeVal = inputEvent.getKeyChar();
```

When you use an event, you can use any of the event's methods, and through the power of inheritance, you can also use methods that belong to any superclass of the event. For example,

any KeyEvent has access to the InputEvent, ComponentEvent, AWTEvent, EventObject, and Object methods, as well as to the KeyEvent methods.

Understanding x- and y-Coordinates

Table 15-5 refers to x- and y-coordinates of a mouse pointer. A window or frame consists of a number of horizontal and vertical pixels on the screen. Any component you place on the screen has a horizontal, or **x-axis**, position as well as a vertical, or **y-axis**, position in the window. The upper-left corner of any display is position 0, 0. The first, or **x-coordinate**, value increases as you travel from left to right across the window. The second, or **y-coordinate**, value increases as you travel from top to bottom. Figure 15-33 illustrates some screen coordinate positions.

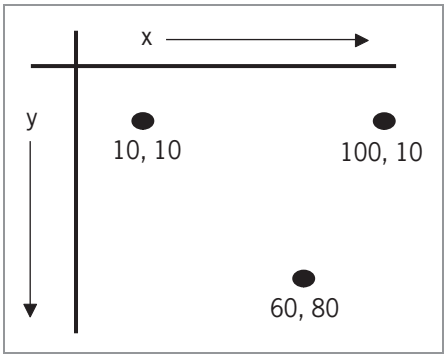

Figure 15-33 Screen coordinate positions

TWO TRUTHS & A LIE

Using AWTEvent Class Methods

1. You use many of the AWTEvent class methods to determine the nature of and facts about an event.
2. The getSource() method returns the Object involved in an event, and the getComponent() method returns the Component involved in an event.
3. The methods isAltDown() and isShiftDown() are ActionEvent methods.

The false statement is #3. The methods isAltDown() and isShiftDown() are KeyEvent methods.

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 button presses; you sometimes simply want to know where a mouse is pointing. 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). The **`MouseInputListener` interface** implements all the methods in both the `MouseListener` and `MouseMotionListener` interfaces; although it has no methods of its own, it is a convenience when you want to handle many different types of mouse events. Tables 15-6 and 15-7 show the methods of the `MouseListener` and `MouseMotionListener` classes, respectively.

Method	Description
`void mouseClicked(MouseEvent e)`	Invoked when the mouse button has been clicked (pressed and released) on a component
`void mouseEntered(MouseEvent e)`	Invoked when the mouse pointer enters a component
`void mouseExited(MouseEvent e)`	Invoked when the mouse pointer exits a component
`void mousePressed(MouseEvent e)`	Invoked when a mouse button has been pressed on a component
`void mouseReleased(MouseEvent e)`	Invoked when a mouse button has been released on a component

Table 15-6 `MouseListener` methods

Many of the methods in Tables 15-6 and 15-7 also appear in tables earlier in this chapter. They are organized by interface here so you can better understand the scope of methods that are available for mouse actions. Don't forget that because `MouseListener`, `MouseMotionListener`, and `MouseInputListener` are interfaces, you must include each method in every program that implements them, even if you choose to place no instructions within some of the methods.

Method	Description
`void mouseDragged(MouseEvent e)`	Invoked when a mouse button is pressed on a component and then dragged
`void mouseMoved(MouseEvent e)`	Invoked when the mouse pointer has been moved onto a component but no buttons have been pressed

Table 15-7 `MouseMotionListener` methods

The `MouseWheelListener` interface contains just one method named `mouseWheelMoved()`, and it accepts a `MouseWheelEvent` argument.

Each of the methods in Tables 15-6 and 15-7 accepts a `MouseEvent` argument. A **`MouseEvent`** is the type of event generated by mouse manipulation. Figure 15-34 shows the inheritance hierarchy of the `MouseEvent` class. From this diagram, you can see that a `MouseEvent` is a type of `InputEvent`, which is a type of `ComponentEvent`. The `MouseEvent` class contains many instance methods and fields that are useful in describing mouse-generated events. Table 15-8 lists some of the more useful methods of the `MouseEvent` class, and Table 15-9 lists some fields.

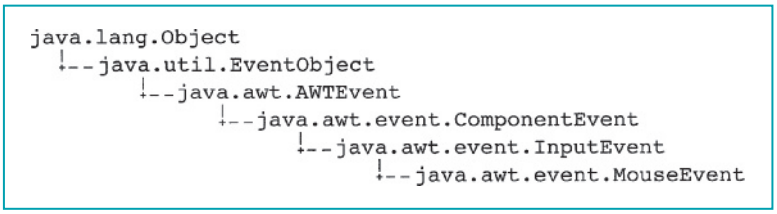

Figure 15-34 The inheritance hierarchy of the `MouseEvent` class

Method	Description
`int getButton()`	Returns which, if any, of the mouse buttons has changed state; uses fields `NOBUTTON`, `BUTTON1`, `BUTTON2`, and `BUTTON3`
`int getClickCount()`	Returns the number of mouse clicks associated with the current event
`int getX()`	Returns the horizontal x-position of the event relative to the source component
`int getY()`	Returns the vertical y-position of the event relative to the source component

Table 15-8 Some useful `MouseEvent` methods

Field	Description
`static int BUTTON1`	Indicates mouse button #1; used by `getButton()`
`static int BUTTON2`	Indicates mouse button #2; used by `getButton()`
`static int BUTTON3`	Indicates mouse button #3; used by `getButton()`
`static int NOBUTTON`	Indicates no mouse buttons; used by `getButton()`
`static int MOUSE_CLICKED`	The "mouse clicked" event
`static int MOUSE_DRAGGED`	The "mouse dragged" event
`static int MOUSE_ENTERED`	The "mouse entered" event
`static int MOUSE_EXITED`	The "mouse exited" event

Table 15-9 Some useful `MouseEvent` fields

Figure 15-35 shows a `JMouseActionFrame` application that demonstrates several of the mouse listener and event methods. `JMouseActionFrame` extends `JFrame`, and because it implements the `MouseListener` interface, it must include all five methods—`mouseClicked()`, `mouseEntered()`, `mouseExited()`, `mousePressed()`, and `mouseReleased()`—even though no actions are included in the `mousePressed()` or `mouseReleased()` methods.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMouseActionFrame extends JFrame implements MouseListener
{
   private int x, y;
   private JLabel label= new JLabel("Do something with the mouse");
   String msg = "";

   public JMouseActionFrame()
   {
      setTitle("Mouse Actions");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      addMouseListener(this);
      add(label);
   }
```

Figure 15-35 The `JMouseActionFrame` application *(continues)*

(*continued*)

```
    public void mouseClicked(MouseEvent e)
    {
       int whichButton = e.getButton();
       msg = "You pressed mouse ";
       if(whichButton == MouseEvent.BUTTON1)
          msg += "button 1.";
       else
          if(whichButton == MouseEvent.BUTTON2)
             msg += "button 2.";
          else
             msg += "button 3.";
       msg += " You are at position " +
          e.getX() + ", " + e.getY() + ".";
       if(e.getClickCount() == 2)
          msg += " You double-clicked.";
       else
          msg += " You single-clicked.";
       label.setText(msg);
    }
    public void mouseEntered(MouseEvent e)
    {
        msg = "You entered the frame.";
        label.setText(msg);
    }
    public void mouseExited(MouseEvent e)
    {
        msg = "You exited the frame.";
        label.setText(msg);
    }

    public void mousePressed(MouseEvent e)
    {
    }

    public void mouseReleased(MouseEvent e)
    {
    }
    public static void main(String[] args)
    {
       JMouseActionFrame mFrame = new JMouseActionFrame();
       final int WIDTH = 750;
       final int HEIGHT = 300;
       mFrame.setSize(WIDTH, HEIGHT);
       mFrame.setVisible(true);
    }
}
```

Figure 15-35 The `JMouseActionFrame` application

The `JMouseActionFrame` application in Figure 15-35 displays messages as the user generates mouse actions. At the start of the class, two integers are declared to hold the mouse position x- and y-coordinates. A `JLabel` and a `String` are also declared to hold messages that inform the user of the mouse actions taken. In the first shaded section of Figure 15-35, the constructor sets a frame title by passing it to the parent of `JMouseActionFrame`, sets a close operation, sets the layout manager, enables the frame to listen for mouse events, and adds the `JLabel` to the `JFrame`.

In Figure 15-35, most of the action occurs in the `mouseClicked()` method (the second unshaded area in the figure). The method builds a `String` that is ultimately assigned to the `JLabel`. The same actions could have been placed in the `mousePressed()` or `mouseReleased()` method because the statements could be placed in the frame just as well at either of those times. Within the `mouseClicked()` method, the `MouseEvent` object named `e` is used several times. It is used with the `getButton()` method to determine which mouse button the user clicked, `getX()` and `getY()` are used to retrieve the mouse position, and `getClickCount()` is used to distinguish between single- and double-clicks.

In Figure 15-35, different messages also are generated in the `mouseEntered()` and `mouseExited()` methods, so the user is notified when the mouse pointer has "entered"—that is, passed over the surface area of—the `JFrame`, the component that is listening for actions.

The `main()` method at the end of the class creates one instance of the `JMouseActionFrame` class and sets its size and visibility.

Figure 15-36 shows the `JMouseActionFrame` application during execution. At this point, the user has just clicked the left mouse button near the upper-right corner of the frame. Of course, in your own applications you might not want only to notify users of their mouse actions; instead, you might want to perform calculations, create files, or generate any other programming tasks.

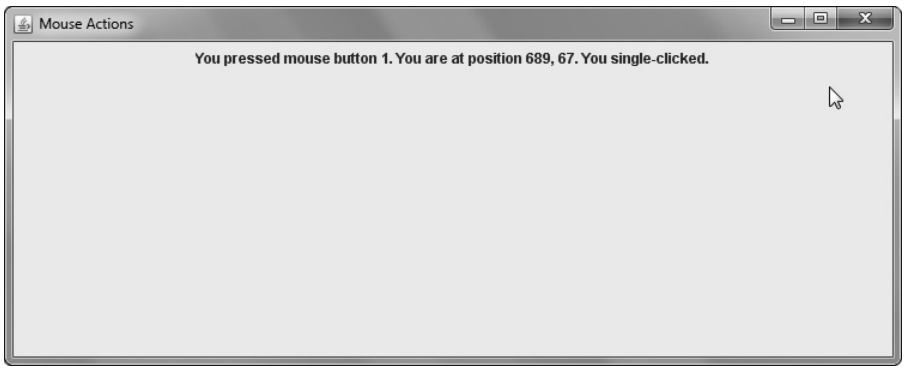

Figure 15-36 Typical execution of the `JMouseActionFrame` application

TWO TRUTHS & A LIE

Handling Mouse Events

1. The `MouseMotionListener` interface provides you with methods that detect the mouse being rolled or dragged across a component surface.
2. The `MouseListener` interface provides you with methods that are analogous to the keyboard event methods `keyPressed()`, `keyTyped()`, and `keyReleased()`.
3. The `MouseListener` interface implements all the methods in the `MouseInputListener` interface.

The false statement is #3. The `MouseInputListener` interface implements all the methods in both the `MouseListener` and `MouseMotionListener` interfaces.

Using Menus

Menus are lists of user options; they are commonly added features in GUI programs. Application users are used to seeing horizontal menu bars across the tops of frames, and they expect to be able to click those options to produce drop-down lists that display more choices. The horizontal list of `JMenus` is a `JMenuBar`. Each `JMenu` can contain options, called `JMenuItems`, or can contain submenus that also are `JMenus`. For example, Figure 15-37 shows a `JFrame` that illustrates the use of the following components:

- A `JMenuBar` that contains two `JMenus` named `File` and `Colors`.
- Three items within the `Colors` `JMenu`: `Bright`, `Dark`, and `White`. `Dark` and `White` are `JMenuItems`. `Bright` is a `JMenu` that holds a submenu. You can tell that `Bright` is a submenu because an arrow sits to the right of its name, and when the mouse hovers over `Bright`, two additional `JMenuItems` appear: `Pink` and `Yellow`.

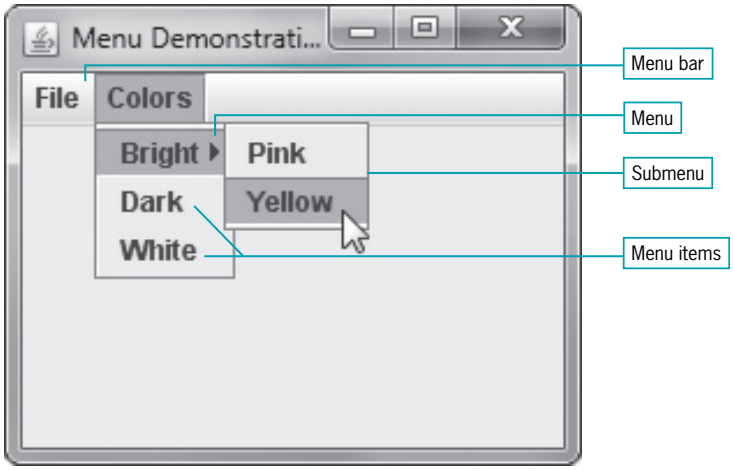

Figure 15-37 A `JFrame` with a horizontal `JMenuBar` that holds two `JMenus`

To create the output shown in Figure 15-37, a series of `JMenuBar`, `JMenu`, and `JMenuItem` objects were created and put together in stages. You can create each of the components you see in the menus in Figure 15-37 as follows:

- You can create a `JMenuBar` much like other objects—by using the `new` operator and a call to the constructor, as follows:

```
JMenuBar mainBar = new JMenuBar();
```

- You can create the two `JMenus` that are part of the `JMenuBar`:

```
JMenu menu1 = new JMenu("File");
JMenu menu2 = new JMenu("Colors");
```

- The three components within the `Colors JMenu` are created as follows:

```
JMenu bright = new JMenu("Bright");
JMenuItem dark = new JMenuItem("Dark");
JMenuItem white = new JMenuItem("White");
```

- The two `JMenuItems` that are part of the `Bright JMenu` are created as follows:

```
JMenuItem pink = new JMenuItem("Pink");
JMenuItem yellow = new JMenuItem("Yellow");
```

Once all the components are created, you assemble them.

- You add the `JMenuBar` to a `JFrame` using the `setJMenuBar()` method as follows:

```
setJMenuBar(mainBar);
```

Using the `setJMenuBar()` method assures that the menu bar is anchored to the top of the frame and looks like a conventional menu bar. Notice that the `JMenuBar` is not added to a `JFrame`'s content pane; it is added to the `JFrame` itself.

- The `JMenus` are added to the `JMenuBar` using the `add()` method. For example:

```
mainBar.add(menu1);
mainBar.add(menu2);
```

- A submenu and two `JMenuItems` are added to the `Colors` menu as follows:

```
menu2.add(bright);
menu2.add(dark);
menu2.add(white);
```

- A submenu can contain its own `JMenuItems`. For example, the `Bright JMenu` that is part of the `Colors` menu in Figure 15-37 contains its own two `JMenuItem` objects:

```
bright.add(pink);
bright.add(yellow);
```

Figure 15-38 shows a complete working program that creates a frame with a greeting and the `JMenu` shown in Figure 15-37.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.Color;
public class JMenuFrame extends JFrame implements
   ActionListener
{
   private JMenuBar mainBar = new JMenuBar();
   private JMenu menu1 = new JMenu("File");
   private JMenu menu2 = new JMenu("Colors");
   private JMenuItem exit = new JMenuItem("Exit");
   private JMenu bright = new JMenu("Bright");
   private JMenuItem dark = new JMenuItem("Dark");
   private JMenuItem white = new JMenuItem("White");
   private JMenuItem pink = new JMenuItem("Pink");
   private JMenuItem yellow = new JMenuItem("Yellow");
   private JLabel label = new JLabel("Hello");
```

Figure 15-38 The `JMenuFrame` class *(continues)*

(*continued*)


```
    public JMenuFrame()
    {
       setTitle("Menu Demonstration");
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       setLayout(new FlowLayout());
       setJMenuBar(mainBar);
       mainBar.add(menu1);
       mainBar.add(menu2);
       menu1.add(exit);
       menu2.add(bright);
       menu2.add(dark);
       menu2.add(white);
       bright.add(pink);
       bright.add(yellow);
       exit.addActionListener(this);
       dark.addActionListener(this);
       white.addActionListener(this);
       pink.addActionListener(this);
       yellow.addActionListener(this);
       add(label);
       label.setFont(new Font("Arial", Font.BOLD, 26));
    }
    public void actionPerformed(ActionEvent e)
    {
       Object source = e.getSource();
       Container con = getContentPane();
       if(source == exit)
          System.exit(0);
       else if(source == dark)
          con.setBackground(Color.BLACK);
       else if(source == white)
          con.setBackground(Color.WHITE);
       else if(source == pink)
          con.setBackground(Color.PINK);
       else con.setBackground(Color.YELLOW);
    }

    public static void main(String[] args)
    {
       JMenuFrame mFrame = new JMenuFrame();
       final int WIDTH = 250;
       final int HEIGHT = 200;
       mFrame.setSize(WIDTH, HEIGHT);
       mFrame.setVisible(true);
    }
}
```

Figure 15-38 The `JMenuFrame` class

In the application in Figure 15-38, each `JMenuItem` becomes a source for an `ActionEvent`, and the `JFrame` is assigned the role of listener for each. The `actionPerformed()` method determines the source of any generated event. If the user selects the Exit option from the `File` menu, the application ends. If the user selects any of the colors from the `Colors` menu, the background color of the `JFrame` is altered accordingly.

Using `JCheckBoxMenuItem` and `JRadioButtonMenuItem` Objects

The `JCheckBoxMenuItem` and `JRadioButtonMenuItem` classes derive from the `JMenuItem` class. Each provides more specific menu items as follows:

- `JCheckBoxMenuItem` objects appear with a check box next to them. An item can be selected (displaying a check mark in the box) or not. Usually, you use check box items to turn options on or off.
- `JRadioButtonMenuItem` objects appear with a round radio button next to them. Users usually expect radio buttons to be mutually exclusive, so you usually make radio buttons part of a `ButtonGroup`. Then, when any radio button is selected, the others are all deselected.

The state of a `JCheckBoxMenuItem` or `JRadioButtonMenuItem` can be determined with the `isSelected()` method, and you can alter the state of the check box with the `setSelected()` method.

Figure 15-39 shows a `JMenuFrame2` application in which two `JCheckBoxMenuItems` and three `JRadioButtonMenuItems` have been added to a `JMenu`. The controls have not yet been assigned any tasks, but Figure 15-40 shows how the menu looks when the application executes.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMenuFrame2 extends JFrame
{
   private JMenuBar mainBar = new JMenuBar();
   private JMenu menu1 = new JMenu("File");
   private JCheckBoxMenuItem check1 = new
      JCheckBoxMenuItem("Check box A");
   private JCheckBoxMenuItem check2 = new
      JCheckBoxMenuItem("Check box B");
```

Figure 15-39 The `JMenuFrame2` application *(continues)*

(*continued*)


```
    private JRadioButtonMenuItem radio1 = new
       JRadioButtonMenuItem("Radio option 1");
    private JRadioButtonMenuItem radio2 = new
       JRadioButtonMenuItem("Radio option 2");
    private JRadioButtonMenuItem radio3 = new
       JRadioButtonMenuItem("Radio option 3");
    private ButtonGroup group = new ButtonGroup();

    public JMenuFrame2()
    {
       setTitle("Menu Demonstration");
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       setLayout(new FlowLayout());
       setJMenuBar(mainBar);
       mainBar.add(menu1);
       menu1.add(check1);
       menu1.add(check2);
       menu1.addSeparator();
       menu1.add(radio1);
       menu1.add(radio2);
       menu1.add(radio3);
       group.add(radio1);
       group.add(radio2);
       group.add(radio3);
    }

    public static void main(String[] args)
    {
       JMenuFrame2 frame = new JMenuFrame2();
       final int WIDTH = 150;
       final int HEIGHT = 200;
       frame.setSize(WIDTH, HEIGHT);
       frame.setVisible(true);
    }
}
```

Figure 15-39 The `JMenuFrame2` application

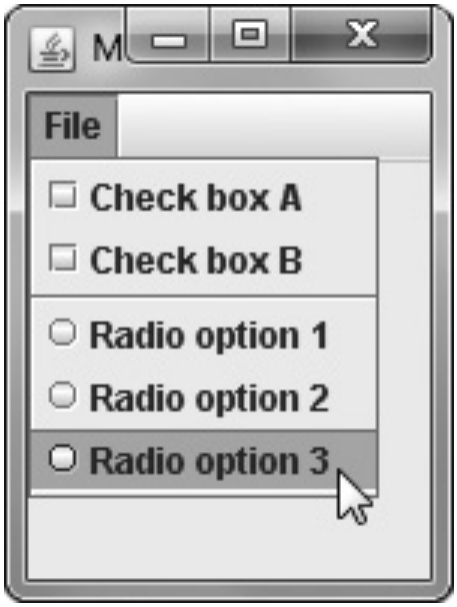

Figure 15-40 Execution of the `JMenuFrame2` application

Using `addSeparator()`

The shaded statement in Figure 15-39 calls the `addSeparator()` method. This method adds a horizontal line to menus in order to visually separate groups for your users. In Figure 15-40, you can see that the separator falls between the `JCheckBoxMenuItems` and the `JRadioButtonMenuItems` because that's the order in which the shaded `addSeparator()` method call was made. The separator does not change the functionality of the menu; it simply makes the menu more visually organized for the user.

Using `setMnemonic()`

A **mnemonic** is a key that causes an already visible menu item to be chosen. You can use the `setMnemonic()` method to provide a shortcut menu key for any visible menu item. For example, when you add the following statement to the `JMenuFrame2` constructor in Figure 15-39, the menu appears as in Figure 15-41:

```
menu1.setMnemonic('F');
```

The mnemonic for the File menu is set to *F*, so the *F* in *File* is underlined. When a user presses Alt+F on the keyboard, the result is the same as if the user had clicked File on the menu: the menu list is opened and displayed.

Figure 15-41 The File menu with a mnemonic applied

Your downloadable student files contain a `JMenuFrame3` application that includes the `setMnemonic()` instruction that produces the output in Figure 15-41.

You should use a different mnemonic for each menu item that has one; if you use the same mnemonic multiple times, only the first assignment works. Usually, you use the first letter of the option—for example, *F* for *File*. If multiple menu items start with the same letter, the convention is to choose the next most prominent letter in the name. For example, *X* is often chosen as the mnemonic for *Exit*.

An **accelerator** is similar to a mnemonic. It is a key combination that causes a menu item to be chosen whether or not it is visible. For example, many word-processing programs allow you to press Ctrl+P to print from anywhere in the program. Only **leaf menu items**—menus that don't bring up other menus—can have accelerators. (They are called "leaves" because they are at the end of a branch with no more branches extending from them.) See the Java Web site for more details.

TWO TRUTHS & A LIE

Using Menus

1. The horizontal list of `JMenus` at the top of a `JFrame` is a `JMenu`.
2. Each `JMenu` can contain options, called `JMenuItems`, or it can contain submenus that also are `JMenus`.
3. You add a `JMenuBar` to a `JFrame` using the `setJMenuBar()` method.

The false statement is #1. The horizontal list of JMenus at the top of a JFrame is a JMenuBar.

You Do It

Using a Menu Bar and JPanels

Next, you create an application for a party planning company that uses a menu bar with multiple user options, and that uses separate `JPanels` with different layout managers to organize components.

1. Open a new file in your text editor, and enter the following first few lines of the `EventSelector` class. The class extends `JFrame` and implements `ActionListener` because the `JFrame` contains potential user mouse selections.

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

2. Create a `JMenuBar` and its two `JMenus` as follows:

```
private JMenuBar mainBar = new JMenuBar();
private JMenu menu1 = new JMenu("File");
private JMenu menu2 = new JMenu("Event types");
```

(continues)

(continued)

3. Next, create the items that will appear within the menus. The File menu contains an Exit option. The Event types menu contains two submenus: Adult and Child. Each of those submenus contains more options. For example, Figure 15-42 shows the expanded Adult event types menu in the finished program.

```
private JMenuItem exit = new JMenuItem("Exit");
private JMenu adult = new JMenu("Adult");
private JMenu child = new JMenu("Child");
private JMenuItem adultBirthday = new JMenuItem("Birthday");
private JMenuItem anniversary = new JMenuItem("Anniversary");
private JMenuItem retirement = new JMenuItem("Retirement");
private JMenuItem adultOther = new JMenuItem("Other");
private JMenuItem childBirthday = new JMenuItem("Birthday");
private JMenuItem childOther = new JMenuItem("Other");
```

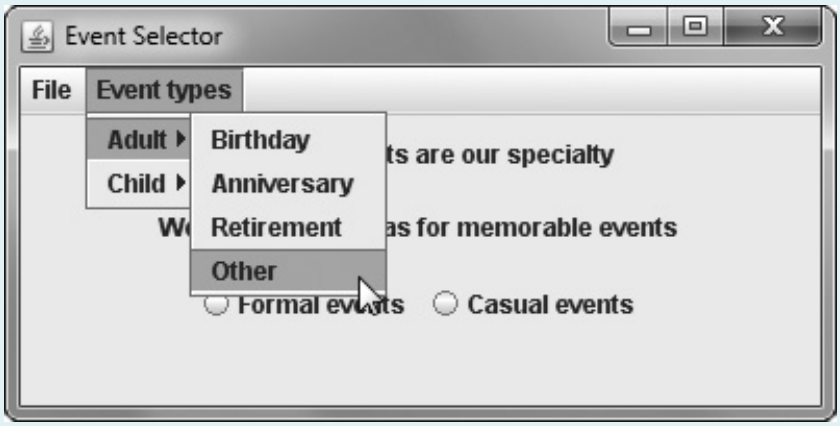

Figure 15-42 The Adult menu

4. Declare several other components that will be used to show how `JFrame`s are composed:

```
private JPanel birthdayPanel = new JPanel();
private JPanel otherPanel = new JPanel();
private JLabel birthdayLabel = new
   JLabel("Birthday events are our specialty");
private JLabel otherLabel = new
   JLabel("We have lots of ideas for memorable events");
private JPanel buttonPanel = new JPanel();
private JRadioButton radButton1 = new
   JRadioButton("Formal events");
private JRadioButton radButton2 = new
   JRadioButton("Casual events");
```

(continues)

(continued)

5. Write the constructor for the `JFrame`. Set the title, the default close operation, and the layout. Call separate methods to compose the menu, to add the necessary action listeners to the menu items, and to lay out the `JFrame`'s components. These tasks could be performed directly within the constructor, but you can place them in separate methods to better organize the application.

```
public EventSelector()
{
   setTitle("Event Selector");
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setLayout(new FlowLayout());
   composeMenus();
   addActionListeners();
   layoutComponents();
}
```

6. Add the `composeMenus()` method. Set the main menu bar, and add two menus to it. Then add one option to the first menu and two submenus to the second menu. Finally, add four items to the first submenu and two items to the other one.

```
public void composeMenus()
{
   setJMenuBar(mainBar);
   mainBar.add(menu1);
   mainBar.add(menu2);
   menu1.add(exit);
   menu2.add(adult);
   menu2.add(child);
   adult.add(adultBirthday);
   adult.add(anniversary);
   adult.add(retirement);
   adult.add(adultOther);
   child.add(childBirthday);
   child.add(childOther);
}
```

7. Add the `addActionListeners()` method, which makes the `JFrame` become a listener for each menu item:

```
public void addActionListeners()
{
   exit.addActionListener(this);
   adultBirthday.addActionListener(this);
   anniversary.addActionListener(this);
   retirement.addActionListener(this);
   adultOther.addActionListener(this);
   childBirthday.addActionListener(this);
   childOther.addActionListener(this);
}
```

(continues)

(continued)

8. The `layoutComponents()` method arranges all the components that appear in the content pane. The `birthdayPanel` object contains a single label. The `otherPanel` object contains a label and another panel (`buttonPanel`) in a grid. The `buttonPanel` contains two radio buttons. For this demonstration, the radio buttons are not functional, but in a more complicated application, an `addActionListener()` method could be applied to them. Also, in a more complicated application, you could continue to place panels within another panel to achieve complex designs.

```
public void layoutComponents()
{
   birthdayPanel.setLayout(new FlowLayout());
   otherPanel.setLayout(new GridLayout(2, 1, 3, 3));
   birthdayPanel.add(birthdayLabel);
   otherPanel.add("other", otherLabel);
   otherPanel.add("buttons", buttonPanel);
   buttonPanel.add(radButton1);
   buttonPanel.add(radButton2);
   add(birthdayPanel);
   add(otherPanel);
}
```

9. Add an `actionPerformed()` method that responds to menu selections. Different background colors are set depending on the user's choices.

```
public void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   Container con = getContentPane();
   if(source == exit)
      System.exit(0);
   else if(source == childBirthday || source == childOther)
      con.setBackground(Color.PINK);
   else
      con.setBackground(Color.WHITE);
   if(source == adultBirthday || source == childBirthday)
   {
      birthdayPanel.setBackground(Color.YELLOW);
      otherPanel.setBackground(Color.WHITE);
   }
   else
   {
      birthdayPanel.setBackground(Color.WHITE);
      otherPanel.setBackground(Color.YELLOW);
   }
}
```

(continues)

(continued)

10. Add the `main()` method, which instantiates an `EventSelector` object and sets its size and visibility. Add a closing curly brace for the class.

```
    public static void main(String[] args)
    {
       EventSelector frame = new EventSelector();
       final int WIDTH = 400;
       final int HEIGHT = 200;
       frame.setSize(WIDTH, HEIGHT);
       frame.setVisible(true);
    }
}
```

11. Save the application as **EventSelector.java**, and then compile and run it. Make various selections and observe the effects. Figure 15-43 shows the running application after the user has made some selections. After you experiment with the application, dismiss the frame.

Figure 15-43 The `EventSelector` application

12. Experiment by making changes to the `EventSelector` application. For example, some menu selections could change the `JFrame` background to a different color, and others could add a new `JLabel` to the `JFrame` content pane.

Don't Do It

- Don't forget that the content pane is operating behind the scenes when you use a top-level container and that, depending on the operations you want to perform, you might need to get a reference to it.
- Don't forget that when you create a custom `Color` object, 0 represents the darkest shade and 255 represents the lightest.
- Don't forget to set a layout manager if you do not want to use the default one for a container.
- Don't use `add()` to place a `JFrame`'s menu bar. You must use the `setMenuBar()` method to place a menu bar correctly.
- Don't use the same mnemonic for multiple menu items.

Key Terms

A **top-level container** is one at the top of a containment hierarchy. The Java top-level containers are `JFrame`, `JDialog`, and `JApplet`.

A **containment hierarchy** is a tree of components that has a top-level container as its root (that is, at its uppermost level).

A **content pane** contains all the visible components in a top-level container's user interface.

A **menu bar** is a horizontal strip that is placed at the top of a container and that contains user options.

A **glass pane** resides above the content pane in a container. It can contain tool tips.

The **`getContentPane()` method** returns a reference to a container's content pane.

The **`Color` class** defines colors for you to use in your applications.

The **`BorderLayout` manager** is the default manager class for all content panes. With the `BorderLayout` manager, components fill the screen in five regions: north, south, east, west, and center.

The **`FlowLayout` manager** arranges components in rows across the width of a `Container`; when the current row is filled, additional `Components` are placed in new rows.

The **preferred size** of a `Component` is its default size.

The **`GridLayout` manager** divides a container surface into a grid.

The **`CardLayout` manager** generates a stack of containers or components, one on top of another.

The **`GridBagLayout` manager** 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` manager** allows multiple components to be laid out either vertically or horizontally. The components do not wrap, so a vertical arrangement of components, for example, stays vertically arranged when the frame is resized.

A **`JPanel`** is a plain, borderless surface that can hold lightweight UI components.

Double buffering is the default buffering strategy in which `JPanels` are drawn offscreen when they are updated and displayed only when complete.

The **primary surface** is the visible screen surface during double buffering.

The **back buffer** is the offscreen image during double buffering.

A **block line transfer**, or **blitting**, is the act of copying the contents from one surface to another.

A **`JScrollPane`** provides scroll bars along the side or bottom of a pane, or both, so that the user can scroll initially invisible parts of the pane into view.

The **viewport** is the viewable area in a `JScrollPane`.

An **adapter class** implements all the methods in an interface, providing an empty body for each method.

The **`KeyListener` interface** contains methods that react to actions the user initiates from the keyboard.

An **action key** is a keyboard key that does not generate a character.

Virtual key codes represent keyboard keys that have been pressed.

The **x-axis** is an imaginary horizontal line that indicates screen position.

The **y-axis** is an imaginary vertical line that indicates screen position.

The **x-coordinate** is a value that increases as you travel from left to right across a window.

The **y-coordinate** is a value that increases as you travel from top to bottom across a window.

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()`.

The **`MouseInputListener` interface** implements all the methods in both the `MouseListener` and `MouseMotionListener` interfaces.

A `MouseEvent` is the type of event generated by mouse manipulation.

Menus are lists of user options.

A **mnemonic** is a key that causes an already visible menu item to be chosen.

An **accelerator** is a key combination that causes a menu item to be chosen, whether or not the menu item is visible.

A **leaf menu item** is a menu item that does not bring up another menu.

Chapter Summary

- Every top-level container has a content pane that contains all the visible components in the container's user interface. The content pane can contain components and other containers. Whenever you create a top-level container, you can get a reference to its content pane using the `getContentPane()` method.
- The `Color` class defines 13 colors for you to use in your applications. It can be used with the `setBackground()` and `setForeground()` methods of the `Component` class to make your applications more attractive and interesting. You also can create more than 16 million custom colors.
- The layout manager assigned to a `Container` determines how the components are sized and positioned within it. The `BorderLayout` manager is the default manager class for all content panes; when you use it, the components fill the screen in five regions: north, south, east, west, and center. The `FlowLayout` manager arranges components in rows across the width of a `Container`. 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. The `CardLayout` manager generates a stack of containers or components, one on top of another.
- A `JPanel` is a plain, borderless surface that can hold lightweight UI components.
- A `JScrollPane` provides scroll bars along the side or bottom of a pane, or both, so that the user can scroll initially invisible parts of the pane into view.
- `ActionEvents` are generated by components that users can click, such as `JButtons` and `JCheckBoxes`, and `TextEvents` are generated by components into which the user enters text, such as a `JTextField`. `MouseEvents` include determining the location of the mouse pointer and distinguishing between a single- and double-click. When you want to listen for an event, you implement an appropriate interface for your class. For every event class, such as `<name>Event`, there is a similarly named `<name>Listener` interface. Instead of implementing a listener class, you can extend an adapter class.
- In addition to the handler methods included with the event listener interfaces, the `AWTEvent` classes themselves contain methods that return information about an event.

- 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()`. 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). The `MouseInputListener` interface implements all the methods in both the `MouseListener` and `MouseMotionListener` interfaces.
- Menus are lists of user options. You use `JMenuBar`, `JMenu`, `JMenuItem`, and other classes in menu creation.

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 of the following 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 _____, 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 from left to right across a `JFrame` surface until the first row is full, at which point a second row is started at the frame 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 from 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` or `JFrame`, the action generates a(n) _____.

    a. `ActionEvent`
    b. `MouseEvent`
    c. `PanelEvent`
    d. `KeyboardEvent`

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` in which an `ActionEvent` originates.

    a. `getObject()`
    b. `getEvent()`
    c. `getOrigin()`
    d. `getSource()`

20. Which of the following is true in a standard menu application?

    a. A `JMenuItem` holds a `JMenu`.
    b. A `JMenuItem` holds a `JMenuBar`.
    c. A `JMenuBar` holds a `JMenu`.
    d. A `JMenu` holds a `JMenuBar`.

Exercises

Programming Exercises

1. Create a `JFrame`, and set the layout to `BorderLayout`. Place a `JButton` in each region, and place the name of an appropriate United States landmark on each `JButton`. For example, New York's Statue of Liberty might be the landmark in the east region. Save the file as **JLandmarkFrame.java**.
2. Create an educational program for children that distinguishes between vowels and consonants as the user clicks buttons. Create 26 `JButtons`, each labeled with a different letter of the alphabet. Create a `JFrame` to hold three `JPanels` in a two-by-two grid. Randomly select eight of the 26 `JButtons` and place four in each of the first two `JPanels`. Add a `JLabel` to the third `JPanel`. When the user clicks a `JButton`, the text of the `JLabel` identifies the button's letter as a vowel or consonant, and then a new randomly selected letter replaces the letter on the `JButton`. Save the file as **JVowelConsonant.java**.
3. Create a `JFrame` that holds five buttons with the names of five different fonts. Include a sixth button that the user can click to make a font larger or smaller. Display a demonstration `JLabel` using the font and size that the user selects. Save the file as **JFontSelector.java**.
4. 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 other regions. Save the file as **JColorFrame.java**.
5. 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 file as **JMovingFrame.java**.
6. 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 an application named `JTeamColors` with `GridLayout`. Display four `JPanelOptions JPanels` that show the names, in their team colors, of four of your favorite sports teams. Save the files as **JPanelOptions.java** and **JTeamColors.java**.
7. Write an application that lets you determine the integer value returned by the `InputEvent` method `getModifiers()` when you click your left, right, or—if you have one—middle mouse button on a `JFrame`. Save the file as **JLeftOrRight.java**.

8. a. Search the Java Web site for information on how to use a `JTextArea`. Write an application for the WebBuy Company that allows a user to compose the three parts of a complete e-mail message: the "To:", "Subject:", and "Message:" text. The "To:" and "Subject:" text areas should provide a single line for data entry. The "Message:" area should allow multiple lines of input and be able to scroll if necessary to accommodate a long message. The user clicks a button to send the e-mail message. When the message is complete and the Send button is clicked, the application should display "Mail has been sent!" on a new line in the message area. Save the file as **JEMail.java**.

   b. Modify the `JEMail` application to include a Clear button that the user can click at any time to clear the "To:", "Subject:", and "Message:" fields. Save the file as **JEMail2.java**.

9. a. Create an application that uses a graphic interface to capture room assignment data for dormitory residents and writes that data to a random access output file. The data required for each assignment includes a room number from 1 through 99 inclusive and the first and last names of the resident. Allow the user to enter data one record at a time and to click a button to save it. Save the class as **CreateRandomDormFile.java**.

   b. Create an application that allows the user to enter a room number and display the name of the stored resident for the room, if any. Save the file as **ReadRandomDormFile.java**.

10. Create a `JFrame` for the Summervale Resort. Allow the user to view information about different rooms available, dining options, and activities offered. Include at least two options in each menu, and display appropriate information when the user makes a choice. Save the file as **SummervaleResort.java**.

Debugging Exercises

1. Each of the following files in the Chapter15 folder of your downloadable student files has syntax and/or logic 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, DebugFifteen1.java will become FixDebugFifteen1.java.

   a. DebugFifteen1.java
   b. DebugFifteen2.java
   c. DebugFifteen3.java
   d. DebugFifteen4.java

Game Zone

As you create some of the games in this section, you might find it convenient to add or remove components in a container after construction. Recall from Chapter 14 that in order for the user to see your changes, you might need to call the `validate()`, `invalidate()`, and `repaint()` methods. You will learn more about the `repaint()` method in the next chapter, *Graphics*.

1. a. Create a Mine Field game in which the user attempts to click 10 panels of a grid before hitting the "bomb." Set up a `JFrame` using `BorderLayout`, use the `NORTH` region for a congratulatory message, and use the `CENTER` region for the game. In the `CENTER` region, create a four-by-five grid using `GridLayout` and populate the grid with `JPanel`s. Set the background color for all the `JPanel`s to `Color.BLUE`. Randomly choose one of the panels to be the bomb; the other 19 panels are "safe." Allow the player to click on grids. If the player chooses a safe panel, turn the panel to `Color.WHITE`. If the player chooses the bomb panel, turn the panel to `Color.RED` and turn all the remaining panels white. If the user successfully chooses 10 safe panels before choosing the bomb, display a congratulatory message in the `NORTH` `JFrame` region. Save the game as **MineField.java**.

   b. Improve the Mine Field game by allowing the user to choose a difficulty level before beginning. Place three buttons labeled "Easy", "Intermediate", and "Difficult" in one region of the `JFrame`, and place the game grid and congratulatory message in other regions. Require the user to select a difficulty level before starting the game, and then disable the buttons. If the user chooses "Easy", the user must select only five safe panels to win the game. If the user selects "Intermediate", require 10 safe panels, as in the original game. If the user selects "Difficult", require 15 safe panels. Save the game as **MineField2.java**.

2. a. Create a game that helps new mouse users improve their hand-eye coordination. Within a `JFrame`, display an array of 48 `JPanel`s in a `GridLayout` using eight rows and six columns. Randomly display an *X* on one of the panels. When the user clicks the correct panel (the one displaying the *X*), remove the *X* and display it on a different panel. After the user has successfully "hit" the correct panel 10 times, display a congratulatory message that includes the user's percentage (hits divided by clicks). Save the file as **JCatchTheMouse.java**.

   b. Review how to use the `GregorianCalendar` class from Chapter 4, and then revise the `JCatchTheMouse` game to conclude by displaying the number of seconds it took the user to click all 10 *X*s. When the application starts, create a `GregorianCalendar` object and use the `get(Calendar.SECOND)` and `get(Calendar.MINUTE)` methods with it to get the `SECOND` and `MINUTE` values at the start of the game. When the user has clicked all 10 *X*s, create a second `GregorianCalendar` object and get the `SECOND` and `MINUTE` values at the end of

the game. If the user starts and ends a game during the same minute, then the playing time is simply the difference between the two `SECOND` values. Make sure your application times the game correctly even if the start and stop times do not occur during the same `MINUTE`. Save the file as **JCatchTheMouseTimed.java**.

c. In the `JCatchTheMouseTimed` game described in Game Zone exercise 2b, the timer does not work correctly if the user happens to play when the hour, day, or year changes. Visit the Java Web site to find out how to use the `GregorianCalendar` class method `getTimeInMillis()`, and then modify the game to measure playing time accurately, no matter when the user plays the game. Save the file as **JCatchTheMouseTimed2.java**.

If you were writing a professional timed game, you would test the timer's accuracy regardless of when the user decided to play. For example, if the user played over the midnight hour on New Year's Eve, you would either have to test the game then (which is impractical), or reset your system's clock to simulate New Year's Eve. If you are writing the programs in this book on a school's computer network, you might be blocked by the administrator from changing the date and time. Even if you are working on your own computer, do not attempt to change the date and time unless you understand the impact on other installed applications. For example, your operating system might assume that an installed virus-protection program is expired, or a financial program might indicate that automatically paid bills are overdue.

3. The game Corner the King is played on a checkerboard. To begin, a checker is randomly placed in the bottom row. The player can move one or two squares to the left or upward, and then the computer can move one or two squares left or up. The first to reach the upper-left corner wins. Design a game in which the computer's moves are chosen randomly. When the game ends, display a message that indicates the winner. Save the game as **CornerTheKing.java**.

4. Create a target practice game that allows the user to click moving targets and displays the number of hits in a 10-second period. Create a grid of at least 100 `JPanel`s. Randomly display an *X* on five panels to indicate targets. As the user clicks each *X*, change the label to indicate a hit. When all five *X*s have been hit, randomly display a new set of five targets. Continue with as many sets as the user can hit in 10 seconds. (Use *www.oracle.com/technetwork/java/index.html* to find how to use the `GregorianCalendar` class method `getTimeInMillis()` to calculate the time change.) When the time is up, display a count of the number of targets hit. Save the file as **JTargetPractice.java**.

5. You set up the card game Concentration by placing pairs of cards face down in a grid. The player turns up two cards at a time, exposing their values. If the cards match, they are removed from the grid. If the cards do not match, they are turned back over so their values are hidden again, and the player selects two more cards to expose. Using the knowledge gained by the previously exposed cards, the player attempts to remove all the pairs of cards from play. Create a Java version of this game using a `GridLayout` that is four rows high and five columns wide. Randomly assign two of the numbers 0 through 9 to each of 20 `JPanel`s, and place each of the

20 `JPanel`s in a cell of the grid. Initially, show only "backs" of cards by setting each panel's background to a solid color. When the user clicks a first card, change its color and expose its value. After the user clicks a second card, change its color to the same color as the first exposed card, expose the second card's value, and keep both cards exposed until the user's mouse pointer exits the second card. If the two exposed cards are different, hide the cards again. If the two turned cards match, then "remove" the pair from play by setting their background colors to white. When the user has matched all 20 cards into 10 pairs, display a congratulatory message. Save the game as **JConcentration.java**.

6. Create a Mine Sweeper game by setting up a grid of rows and columns in which "bombs" are randomly hidden. You choose the size and difficulty of the game; for example, you might choose to create a fairly simple game by displaying a four-by-five grid that contains four bombs. If a player clicks a panel in the grid that contains a bomb, then the player loses the game. If the clicked panel is not a bomb, display a number that indicates how many adjacent panels contain a bomb. For example, if a user clicks a panel containing a 0, the user knows it is safe to click any panel above, below, beside, or diagonally adjacent to the cell, because those cells cannot possibly contain a bomb. If the player loses by clicking a bomb, display all the numeric values as well as the bomb positions. If the player succeeds in clicking all the panels except those containing bombs, the player wins and you should display a congratulatory message. Figure 15-44 shows the progression of a typical game. In the first screen, the user has clicked a panel, and the display indicates that two adjacent cells contain a bomb. In the second screen, the user has clicked a second panel, and the display indicates that three adjacent cells contain bombs. In the last screen, the user has clicked a bomb panel, and all the bomb positions are displayed. Save the game as **MineSweeper.java**.

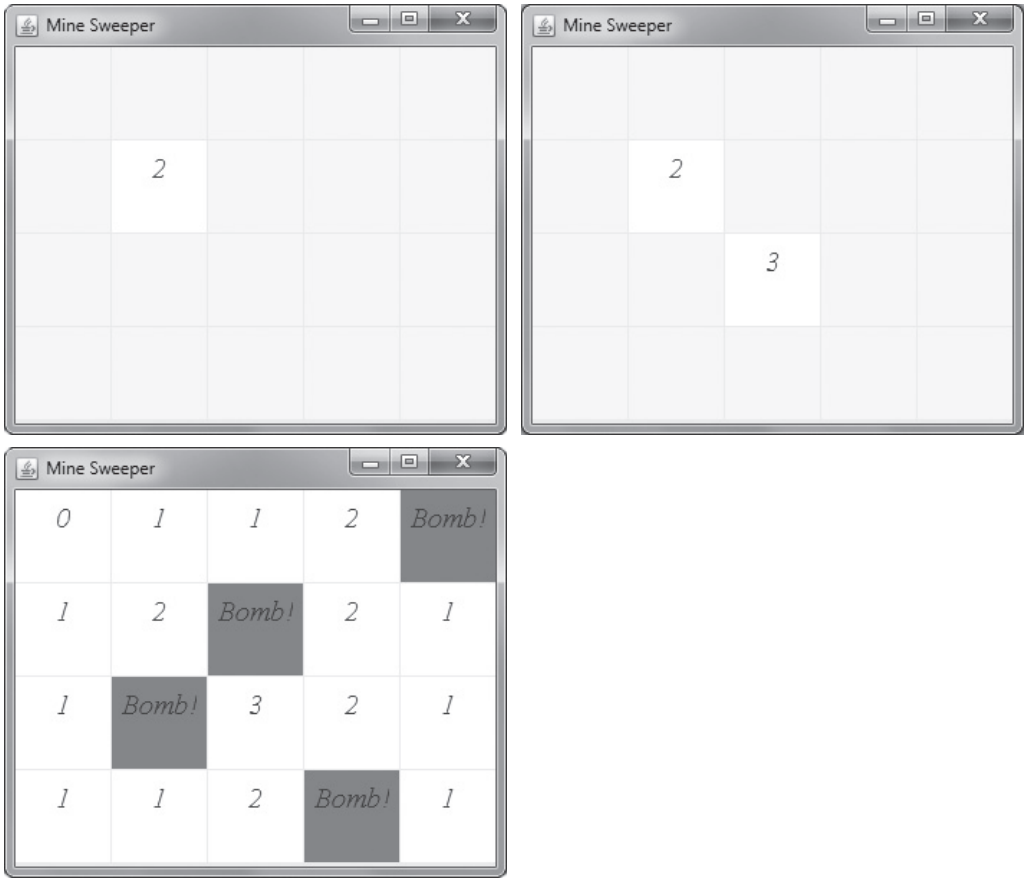

Figure 15-44 Typical progression of MineSweeper game

7. Create the game Lights Out using a `BorderLayout`. Place a five-by-five grid of panels in one region, and reserve another region for a congratulatory message. Randomly set each panel in the grid to a dark color or light color. The object of the game is to force all the panels to be dark, thus turning the "lights out." When the player clicks a panel, turn all the panels in the same row and column, including the clicked panel, to the opposite color. For example, if the user clicks the panel in the second row, third column, then darken all the light-colored panels in the second row and third column, and lighten all the dark-colored panels in that row and column. When all the panels in the grid are dark, all the lights are out, so display a congratulatory message. Save the game as **LightsOut.java**.

8. The game StopGate is played on a checkerboard with a set of dominoes; each domino is large enough to cover two checkerboard squares. One player places a domino horizontally on the checkerboard, covering any two squares. The other player then places a domino vertically to cover any other two squares. When a player has no more moves available, that player loses. Create a computerized version of the game in which the player places the horizontal pieces and the computer randomly selects a position for the vertical pieces. (Game construction will be simpler if you allow the player to select only the left square of a two-square area and assume that the domino covers that position plus the position immediately to the right.) Use a different color for the player's dominoes and the computer's dominoes. Display a message naming the winner when no more moves are possible. Figure 15-45 shows a typical game after the player (blue) and computer (black) have each made one move, and near the end of the game when the player is about to win—the player has two moves remaining, but the computer has none. Save the file as **StopGate.java**.

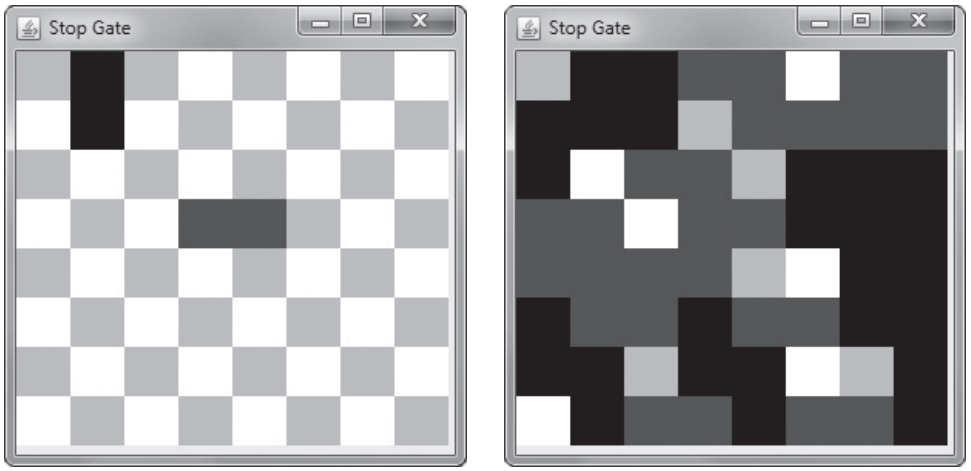

Figure 15-45 A typical game of StopGate just after play begins and near the end of the game

Case Problems

1. In Chapter 14, you created an interactive GUI application for Carly's Catering that allows the user to enter a number of guests for an event and to choose an entrée, two side dishes, and a dessert from groups of choices. Then, the application displays the cost of the event and a list of the chosen items. Now, modify the interface to include separate panels for the guest number entry, each group of menu choices, and the output. Use at least two different layout managers and at least two different colors in your application. Save the program as **JCarlysCatering.java**.
2. In Chapter 14, you created an interactive GUI application for Sammy's Seashore Rentals that allows the user to enter a rental time in hours, an equipment type, and a lesson option. Then, the application displays the cost of the rental and rental details. Now, modify the interface to include separate panels for the hour entry, each group of menu choices, and the output. Use at least two different layout managers and at least two different colors in your application. Save the program as **JSammysSeashore.java**.