

Part III: Graphics Programming

In Part II, "Object-Oriented Programming," you learned the basics of object-oriented programming. The design of the API for Java graphics programming is an excellent example of how the object-oriented principle is applied. In the chapters that follow in this part of the book, you will learn the architecture of Java graphics programming API and use the user interface components to develop graphics applications and applets.

Chapter 10 Getting Started with Graphics
Programming

Chapter 11 Creating User Interfaces

Chapter 12 Applets and Advanced Graphics

Getting Started with Graphics Programming

Objectives

- Describe the Java graphics programming class hierarchy.
- Use frames, panels, and simple UI components.
- Understand the role of layout managers.
- Use the FlowLayout, GridLayout, and BorderLayout managers.
- Become familiar with the paintComponent method.
- Become familiar with the classes Color, Font, and FontMetrics.
- Be able to use the drawing methods in the Graphics class.
- Understand the concept of event-driven programming.
- Become familiar with the Java event-delegation model: event registration, listening, and handling.

Introduction

Until now, you have only used text-based input and output. You used `MyInput.readInt()` and `MyInput.readDouble()` to read numbers from the keyboard, and `System.out.println` to display results on the console. This is the old-fashioned way to program. Today's client/server and Web-based applications use a graphical user interface known as GUI (pronounced goo-ee).

When Java was introduced, the graphics components were bundled in a library known as the Abstract Windows Toolkit (AWT). For every platform on which Java runs, the AWT components are automatically mapped to the platform-specific components through their respective agents, known as *peers*. AWT is fine for developing simple GUI applications, but not for developing comprehensive GUI projects. Besides, AWT is prone to platform-specific bugs because its peer-based approach heavily relies on the underlying platform. With the release of Java 2, the AWT user interface components were replaced by a more robust, versatile, and flexible library known as the *Swing components*. Swing components are painted directly on canvases using Java code, except for components that are subclasses of `java.awt.Window` or `java.awt.Panel`, which must be drawn using native GUI on a specific platform. Swing components are less dependent on the target platform and use less of the native GUI resource. For this reason, Swing components that don't rely on native GUI are referred to as *lightweight components*, and AWT components are referred to as *heavyweight components*. Although AWT components are still supported in Java 2, I recommend that you learn to program using the Swing components, because the AWT user interface components will eventually fade away.

Java provides a rich set of classes to help you build graphical user interfaces. You can use various GUI-building classes—frames, panels, labels, buttons, text fields, text areas, combo boxes, check boxes, radio buttons, menus, scroll bars, scroll panes, and tabbed panes—to construct user interfaces. This chapter introduces the basics of Java graphics programming. Specifically, it discusses GUI components and their relationships, containers and layout managers, colors, fonts, and drawing geometric figures, such as lines, rectangles, ovals, arcs, and polygons, and introduces event-driven programming.

NOTE: The Swing components do not replace all the classes in AWT, only the AWT user interface components (`Button`, `TextField`, `TextArea`, etc.). The AWT helper classes (`Graphics`, `Color`, `Font`, `FontMetrics`, and `LayoutManager`) remain

unchanged. In addition, the Swing components use the AWT event model.

The Java Graphics API

The design of the Java graphics programming API is an excellent example of the use of classes, inheritance, and interfaces. The graphics API contains the essential classes listed below. Their hierarchical relationships are shown in Figure 10.1.

*****Insert Figure 10.1 (Same as Figure 8.1 in the 3rd Edition, p287)**

Figure 10.1

Java graphics programming utilizes the classes shown in this hierarchical diagram.

- **Component**: This is a superclass of all user interface classes.
- **Container**: This is used to group components. A layout manager is used to position and place components in a container in the desired location and style. Frames, panels, and applets are examples of containers.
- **JComponent**: This is a superclass of all the lightweight Swing components, which are drawn directly on canvases using Java code, rather than on specific platforms using native GUI. Its subclasses (JButton, JCheckBox, JMenu, JRadioButton, JLabel, JList, TextField, TextArea, JScrollPane) are the basic elements for constructing the GUI.
- **JFrame**: This is a window not contained inside another window. JFrame is the container that holds other Swing user interface components in Java graphical applications.
- **JDialog**: This is a popup window, or message box generally used as a temporary window to receive additional information from the user or to provide notification that an event has occurred.
- **JApplet**: This is a subclass of Applet. You must extend JApplet to create a Swing-based Java applet.
- **JPanel**: This is an invisible container that holds user interface components. Panels can be nested. You can place panels inside panels and in a frame in Java

applications or in an applet in Java applets. JPanel can also be used as a canvas to draw graphics.

- **Graphics**: This is an abstract class that provides a graphical context for drawing strings, lines, and simple shapes.
- **Color**: This deals with the colors of graphics components. For example, you can specify background or foreground colors in a component like JFrame and JPanel, or you can specify colors of lines, shapes, and strings in drawings.
- **Font**: This is used for drawings in Graphics. For example, you can specify the font type (SansSerif), style (bold), and size (24 points) to draw a string.
- **FontMetrics**: This is an abstract class used to get properties of the fonts used in drawings.

The graphics classes can be classified into three groups: the *container classes*, the *component classes*, and the *helper classes*. The container classes, such as JFrame, JPanel, and JApplet, are used to contain other components. The UI component classes, such as JButton, JTextField, JTextArea, JComboBox, JList, JRadioButton, and JMenu, are subclasses of JComponent. The helper classes, such as Graphics, Color, Font, FontMetrics, Dimension, and LayoutManager, are used by components and containers to draw and place objects.

The JFrame, JApplet, JDialog, and JComponent classes and their subclasses are grouped in the package javax.swing. All the other classes in Figure 10.1 are grouped in the package java.awt. Most Swing components are named with a prefixed J. For example, the Swing version of Button is called JButton to distinguish it from its AWT counterpart.

NOTE: Swing is a comprehensive solution to developing graphic user interfaces. There are over 250 classes in Swing, some of which are illustrated in Figure 10.2. Since the discussion in this book is only an introduction to Java graphics programming using Swing, the components listed in the dotted rectangle are not covered. For a more detailed treatment of Swing components, including model-view architecture, look and feel, and advanced components, please refer to my *Rapid Java Application Development Using JBuilder 4*, published by Prentice-Hall.

TIP: Do not mix Swing user interface components like JButton with AWT user interface components

like Button. For example, do not place JButton in java.awt.Panel, and similarly do not place Button in javax.swing.JPanel. Mixing them may cause problems. This book uses Swing user interfaces exclusively.

*****Insert Figure 10.2 (Same as Figure 8.2 in the 3rd Edition, p289)**

Figure 10.2

JComponent and its subclasses are the basic elements for building graphical user interfaces.

Frames

To create a user interface, you need to create either a frame or an applet to hold the user interface components. Figure 10.3 provides examples of possible user interface layouts in a frame and an applet.

*****Insert Figure 10.3 (Same as Figure 8.3 in the 3rd Edition, p289)**

Figure 10.3

A frame or an applet can contain menus, panels, and user interface components. Panels are used to group user interface components. Panels can contain other panels.

Creating Java applets will be introduced in Chapter 12, "Applets and Advanced Graphics." This section introduces the procedure for creating frames.

Creating a Frame

The following program creates a frame:

```
// MyFrame.java: Display a frame
package chapter10;

import javax.swing.*;

public class MyFrame
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Test Frame");
        frame.setSize(400, 300);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Because JFrame is in the package javax.swing, the statement import javax.swing.* makes available all the classes from the javax.swing package, including JFrame, so that they can be used in the MyFrame class.

The following two constructors are used to create a JFrame object.

- public JFrame();

This constructor creates a JFrame object that is untitled.

- public JFrame(String title);

This creates a JFrame object with a specified title.

The frame is not displayed *until* the frame.setVisible(true) method is applied. frame.setSize(400, 300) specifies that the frame is 400 pixels wide and 300 pixels high. If the setSize method is not used, the frame will be sized at 0 by 0 pixels, and nothing will be seen except the title bar. Since the setSize and setVisible methods are both defined in the Component class, they are inherited by the JFrame class. Later you will see that these methods are also useful in many other subclasses of Component.

When you run the MyFrame program, the following window will be displayed on-screen (see Figure 10.4).

*****Insert Figure 10.4 (Same as Figure 8.4 in the 3rd Edition, 290)**

Figure 10.4

The program creates and displays a frame with the title Test Frame.

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) tells the program to terminate when the frame is closed. If this statement is not used, the program does not terminate when the frame is closed. In that case, you have to stop the program by pressing Ctrl+C at the DOS prompt window in Windows or use the kill command to stop the process in Unix.

Centering a Frame (Optional)

By default, a frame is displayed in the upper-left corner of the screen. To display a frame at a specified location, use the setLocation(x, y) method in the JFrame class. This method places the upper-left corner of a frame at location (x, y).

To center a frame on the screen, you need to know the width and height of the screen and the frame in order to determine the frame's upper-left coordinates. The screen's width and height can be obtained using the java.awt.Toolkit class:

```
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();  
int screenWidth = screenSize.width;  
int screenHeight = screenSize.height;
```

Therefore, as shown in Figure 10.5, the upper left x and y coordinates of the centered frame frame can be:

```
Dimension frameSize = frame.getSize();  
int x = (screenWidth - frameSize.width)/2;  
int y = (screenHeight - frameSize.height)/2;
```

*****Insert Figure 10.5 (Same as Figure 8.5 in the 3rd Edition, p291)**

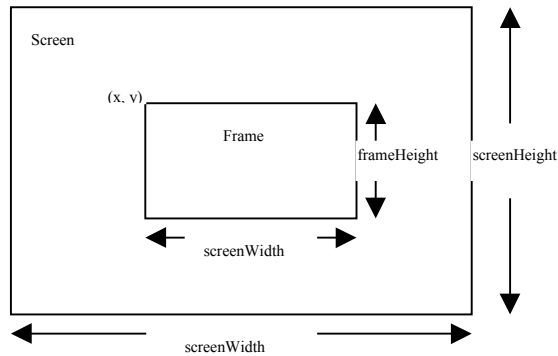


Figure 10.5

The frame is centered on the screen.

The getSize method is defined in the Component class. The java.awt.Dimension class encapsulates the width and height of a component (in integer precision) in a single object. To display a centered frame, the program can be modified as follows:

```
// CenterFrame.java: Display a frame  
package chapter10;  
  
import javax.swing.*;  
import java.awt.*;  
  
public class CenterFrame  
{  
    public static void main(String[] args)  
    {  
        JFrame frame = new JFrame("Centered Frame");  
        frame.setSize(400, 300);  
  
        // New in JDK 1.3 to exit the program upon closing
```



```

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Get the dimension of the screen
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int screenWidth = screenSize.width;
        int screenHeight = screenSize.height;

        // Get the dimension of the frame
        Dimension frameSize = frame.getSize();
        int x = (screenWidth - frameSize.width)/2;
        int y = (screenHeight - frameSize.height)/2;

        frame.setLocation(x, y);
        frame.setVisible(true);
    }
}

```

Adding Components to a Frame

The frame shown in Figure 10.4 is empty. Using the `add` method, you can add components into the frame's content pane, as follows:

```

// MyFrameWithComponents.java: Add components into a frame
package chapter10;

import javax.swing.*;

public class MyFrameWithComponents
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Adding Components into the Frame");

        // Add a button into the frame
        frame.getContentPane().add(new JButton("OK"));

        frame.setSize(400, 300);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

```

The `getContentPane` method in the `JFrame` class returns the content pane of the frame. The content pane holds the components in the frame. An object of `JButton` was created using `new JButton("OK")`, and this object was added to the content pane of the frame. When you run the program `MyFrameWithComponents`, the following window will be displayed in Figure 10.6.

*****Insert Figure 10.6 (Same as Figure 8.6 in the 3rd Edition, p293)**

Figure 10.6

An OK button is added to the frame.

The button is always centered in the frame and occupies the entire frame no matter how you resize the frame in Figure 10.6. This is because the content pane uses a layout manager, which is responsible for laying out components in

the frame. The default layout manager for the content pane places the button in the center of the frame. In the next section, you will use several different layout managers to place the components in the desired locations.

Layout Managers

In many other window systems, the user interface components are arranged by using hard-coded pixel measurements. For example, put a button at location (10, 10) in the window. Using hard-coded pixel measurements, the user interface might look fine on one system but be unusable on another. Java's layout managers provide a level of abstraction that automatically maps your user interface on all window systems.

The Java GUI components are placed in containers, where they are arranged by the container's layout manager. In the preceding program, you did not specify where to place the OK button in the frame, but Java knows where to place the button because the layout manager works behind the scenes to place components in the correct locations. The five basic layout managers are FlowLayout, GridLayout, BorderLayout, CardLayout, and GridBagLayout. These classes implement the LayoutManager interface.

Layout managers are set in containers. The syntax to set a layout manager is as follows:

```
container.setLayout(new SpecificLayout());
```

To add a component to a container, use the add method. To remove a component from a container, use the remove method. The following statements create a button and add it to a container.

```
JButton jbtOK = new JButton("OK");  
container.add(jbtOK);
```

The following statement remove the button from the container:

```
container.remove(jbtOK);
```

The container can be the content pane of a frame or an applet, or an instance of JPanel.

The following sections introduce the FlowLayout, GridLayout, and BorderLayout managers. The CardLayout and GridBagLayout are introduced in Chapter 12.

FlowLayout

FlowLayout is the simplest layout manager. The components are arranged in the container from left to right in the order in which they were added. When one row is filled, a new row is started. You can specify the way the components are aligned by using one of three constants: FlowLayout.RIGHT, FlowLayout.CENTER, or FlowLayout.LEFT. You can also specify the gap between components in pixels. FlowLayout has three constructors:

- `public FlowLayout(int align, int hGap, int vGap)`

This constructs a new FlowLayout with the specified alignment, horizontal gap, and vertical gap. The gaps are the distances in pixels between components.

- `public FlowLayout(int alignment)`

This constructs a new FlowLayout with a specified alignment and a default gap of five pixels horizontally and vertically.

- `public FlowLayout()`

This constructs a new FlowLayout with a default center alignment and a default gap of five pixels horizontally and vertically.

Example 10.1 Testing the FlowLayout Manager

This example enables a program to arrange components in a frame by using the FlowLayout manager with a specified alignment and horizontal and vertical gaps. The program uses the following simple code to arrange 10 buttons in a frame. The output is shown in Figure 10.7.

```
// ShowFlowLayout.java: Demonstrate using FlowLayout
package chapter10;

import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.Container;
import java.awt.FlowLayout;

public class ShowFlowLayout extends JFrame
{
    /**Default constructor*/
    public ShowFlowLayout()
    {
        // Get the content pane of the frame
        Container container = getContentPane();

        // Set FlowLayout, aligned left with horizontal gap 10
        // and vertical gap 20 between components
        container.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));

        // Add buttons to the frame
        for (int i=1; i<=10; i++)
            container.add(new JButton("Component " + i));
    }

    /**Main method*/
    public static void main(String[] args)
```

```

    {
        ShowFlowLayout frame = new ShowFlowLayout();
        frame.setTitle("Show FlowLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
}

```

*****Insert Figure 10.7 (Same as Figure 8.7 in the 3rd Edition, p295)**

Figure 10.7

The components are added with the FlowLayout manager to fill in the rows in the container one after another.

Example Review

This example creates a program using a style different from the programs in the previous section. In the previous section, frames were created using the JFrame class. This example creates a class named ShowFlowLayout that extends the JFrame class. The main method in this program creates an instance of ShowFlowLayout. The constructor of ShowFlowLayout constructs and places the components in the frame. This is a preferred style of creating GUI applications for two reasons: 1. Creating a GUI application is to create a frame, hence, it is natural to define a frame to extend JFrame; 2. The new class could be reused if desirable. Using one style consistently helps students to read the programs. From now on, all the GUI main classes will extend the JFrame class. The constructor of the main class constructs the user interface. The main method creates an instance of the main class and then displays the frame.

The FlowLayout manager is used to place components in the frame in this example. If you resize the frame, the components are automatically rearranged to fit in the frame.

If you replace the setLayout statement with setLayout(new FlowLayout(FlowLayout.LEFT, 0, 0)), you will see all the buttons left-aligned with no gaps.

An anonymous FlowLayout object was created in the statement:

```

container.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));

```

This statement is equivalent to the following code:

```
FlowLayout layout = new FlowLayout(FlowLayout.LEFT, 10, 20);  
container.setLayout(layout);
```

This code creates an explicit reference to the object layout of the FlowLayout class. This explicit reference is not necessary, because the object is not directly referenced in the ShowLayout class.

The setTitle method is defined in the java.awt.Frame class. Since JFrame is a subclass of Frame, you can use it to set a title for an object of JFrame.

CAUTION: Do not forget to put the new operator before LayoutManager when setting a layout style; for example, setLayout(new FlowLayout()).

NOTE: The constructor ShowFlowLayout() does not explicitly invoke the constructor JFrame(), but the constructor JFrame() is invoked implicitly.

GridLayout

The GridLayout manager arranges components in a grid (matrix) formation with the number of rows and columns defined by the constructor. The components are placed in the grid from left to right, starting with the first row, then the second, and so on, in the order in which they are added. The GridLayout manager has three constructors:

- public GridLayout(int rows, int columns, int hGap, int vGap)

This constructs a new GridLayout with the specified number of rows and columns, along with specified horizontal and vertical gaps between components in the container.

- public GridLayout(int rows, int columns)

This constructs a new GridLayout with the specified number of rows and columns. The horizontal and vertical gaps are zero.

- public GridLayout()

This constructs a new GridLayout with one column in a single row.

You can specify the number of rows and columns in the grid. The basic rule is as follows:

- The number of rows or the number of columns can be zero, but not both. If one is zero and the other is

nonzero, the nonzero dimension is fixed, while the zero dimension is determined dynamically by the layout manager. For example, if you specify zero rows and three columns for a grid that has 10 components, GridLayout creates three fixed columns of four rows, with the last row containing one component. If you specify three rows and zero columns for a grid that has 10 components, GridLayout creates three fixed rows of four columns, with the last row containing two components.

- If both the number of rows and the number of columns are nonzero, the number of rows is the dominating parameter; that is, the number of rows is fixed, and the layout manager dynamically calculates the number of columns. For example, if you specify three rows and three columns for a grid that has 10 components, GridLayout creates three fixed rows of four columns, with the last row containing two components.

Example 10.2 Testing the GridLayout Manager

This example presents a program that arranges components in a frame with GridLayout. The program gives the following code to arrange 10 buttons in a grid of four rows and three columns. Its output is shown in Figure 10.8.

```
// ShowGridLayout.java: Demonstrate using GridLayout
package chapter10;

import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.GridLayout;
import java.awt.Container;

public class ShowGridLayout extends JFrame
{
    /**Default constructor*/
    public ShowGridLayout()
    {
        // Get the content pane of the frame
        Container container = getContentPane();

        // Set GridLayout, 4 rows, 3 columns, and gaps 5 between
        // components horizontally and vertically
        container.setLayout(new GridLayout(4, 3, 5, 5));

        // Add buttons to the frame
        for (int i=1; i<=10; i++)
            container.add(new JButton("Component " + i));
    }

    /**Main method*/
    public static void main(String[] args)
    {
        ShowGridLayout frame = new ShowGridLayout();
        frame.setTitle("Show GridLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
}
```

*****Insert Figure 10.8 (Same as Figure 8.8 in the 3rd Edition, p297)**

Figure 10.8

The GridLayout manager divides the container into grids, then the components are added to fill in the cells, row by row.

Example Review

If you resize the frame, the layout of the buttons remains unchanged (that is, the number of rows and columns does not change, and the gaps don't change either).

All components are given equal size in the container of GridLayout.

Replacing the setLayout statement with setLayout(new GridLayout(3, 10)) would yield three rows and four columns, with the last row containing two components. The columns parameter is ignored because the rows parameter is nonzero. The actual number of columns is calculated by the layout manager.

NOTE: In FlowLayout and GridLayout, the order in which the components are added to the container is important. It determines the location of the components in the container.

BorderLayout

The BorderLayout manager divides the window into five areas: East, South, West, North, and Center. Components are added to a BorderLayout by using add(Component, index), where index is a constant BorderLayout.EAST, BorderLayout.SOUTH, BorderLayout.WEST, BorderLayout.NORTH, or BorderLayout.CENTER. You can use one of the following two constructors to create a new BorderLayout:

- public BorderLayout(int hGap, int vGap)

This constructs a new BorderLayout with the specified horizontal and vertical gaps between the components.

- public BorderLayout()

This constructs a new BorderLayout without horizontal or vertical gaps.

The components are laid out according to their preferred sizes and where they are placed in the container. The north and south components can stretch horizontally; the east and

west components can stretch vertically; the center component can stretch both horizontally and vertically to fill any empty space.

Example 10.3 Testing the BorderLayout Manager

The example presents the following code to place East, South, West, North, and Center buttons in the frame by using BorderLayout. The output of the program is shown in Figure 10.9.

```
// ShowBorderLayout.java: Demonstrate using BorderLayout
package chapter10;

import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.Container;
import java.awt.BorderLayout;

public class ShowBorderLayout extends JFrame
{
    /**Default constructor*/
    public ShowBorderLayout()
    {
        // Get the content pane of the frame
        Container container = getContentPane();

        // Set BorderLayout with horizontal gap 5 and vertical gap 10
        container.setLayout(new BorderLayout(5, 10));

        // Add buttons to the frame
        container.add(new JButton("East"), BorderLayout.EAST);
        container.add(new JButton("South"), BorderLayout.SOUTH);
        container.add(new JButton("West"), BorderLayout.WEST);
        container.add(new JButton("North"), BorderLayout.NORTH);
        container.add(new JButton("Center"), BorderLayout.CENTER);
    }

    /**Main method*/
    public static void main(String[] args)
    {
        ShowBorderLayout frame = new ShowBorderLayout();
        frame.setTitle("Show BorderLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

*****Insert Figure 10.9 (Same as Figure 8.9 in the 3rd Edition, p299)**

Figure 10.9

BorderLayout divides the container into five areas, each of which can hold a component.

Example Review

The buttons are added to the frame. Note that the add method for BorderLayout is different from FlowLayout and GridLayout. With BorderLayout you specify where to put the components.

It is unnecessary to place components to occupy all the areas. If you remove the East button from the program and rerun it, you will see that the center stretches rightward to occupy the East area.

NOTE: For convenience, BorderLayout interprets the absence of an index specification as BorderLayout.CENTER. For example, add(component) is the same as add(Component, BorderLayout.CENTER).

TIP: To avoid mistakes, I recommend that you explicitly set a layout style for a container, even though BorderLayout is used by default for the content pane of a JFrame.

Using Panels as Containers

Suppose that you want to place 10 buttons and a text field on a frame. The buttons are placed in grid formation, but the text field is placed on a separate row. It is difficult to achieve the desired look by placing all the components in a single container. With Java graphics programming, you can divide a window into panels. Panels act as smaller containers to group user interface components. You add the buttons in one panel, and then add the panel into the frame. The Swing version of panel is JPanel. The constructor in the JPanel class is JPanel(). To add a button to the panel p, for instance, you can use:

```
JPanel p = new JPanel();  
p.add(new JButton("ButtonName"));
```

By default, JPanel uses FlowLayout. Panels can be placed inside a frame or inside another panel. The following statement places panel p into frame f:

```
f.getContentPane().add(p);
```

NOTE: To add a component to JFrame, you actually add it to the content pane of JFrame. To add a component to a panel, you add it directly to the panel using the add method.

Example 10.4 Testing Panels

This example uses panels to organize components. The program creates a user interface for a microwave oven, as shown in Figure 10.10. The program is given as follows:

```
// TestPanels.java: Use panels to group components  
package chapter10;  
  
import java.awt.*;
```

```

import javax.swing.*;

public class TestPanels extends JFrame
{
    /**Default constructor*/
    public TestPanels()
    {
        // Get the content pane of the frame
        Container container = getContentPane();

        // Set BorderLayout for the frame
        container.setLayout(new BorderLayout());

        // Create panel p1 for the buttons and set GridLayout
        JPanel p1 = new JPanel();
        p1.setLayout(new GridLayout(4, 3));

        // Add buttons to the panel
        for (int i=1; i<=9; i++)
        {
            p1.add(new JButton(" " + i));
        }

        p1.add(new JButton(" " + 0));
        p1.add(new JButton("Start"));
        p1.add(new JButton("Stop"));

        // Create panel p2 to hold a text field and p1
        JPanel p2 = new JPanel();
        p2.setLayout(new BorderLayout());
        p2.add(new JTextField("Time to be displayed here"),
            BorderLayout.NORTH);
        p2.add(p1, BorderLayout.CENTER);

        // Add p2 and a button to the frame
        container.add(p2, BorderLayout.EAST);
        container.add(new JButton("Food to be placed here"),
            BorderLayout.CENTER);
    }

    /**Main method*/
    public static void main(String[] args)
    {
        TestPanels frame = new TestPanels();
        frame.setTitle("The Front View of a Microwave Oven");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 250);
        frame.setVisible(true);
    }
}

```

*****Insert Figure 10.10 (Same as Figure 8.10 in the 3rd Edition, p301)**

Figure 10.10

The program uses panels to organize components.

Example Review

To achieve the desired layout, the program uses panel p1 of the GridLayout to group the number buttons, the Stop button, and the Start button, uses panel p2 of BorderLayout to hold a text field in the north and p1 in the center. The button representing the food is placed in the center of frame, and p2 is placed in the west of the frame.

The statement

```
p2.add(new JTextField("Time to be displayed here"),  
        BorderLayout.NORTH);
```

creates an instance of JTextField. Text field is a GUI component that can be used for user input as well as to display values. Text fields will be introduced in Chapter 11, "Creating User Interfaces."

Drawing Graphics in Panels

Panels are invisible and are used as small containers that group components to achieve a desired layout look. Another important use of JPanel is for drawing strings and graphics. To draw in a panel, you create a new class that extends JPanel and override the paintComponent method to tell the panel how to draw things. You can then display strings, draw geometric shapes, and view images on the panel. Although you can draw things directly in a frame or an applet using the paint method, I recommend that you use JPanel to draw messages and shapes and to show images; this way your drawing does not interfere with other components. The signature of the paintComponent method is as follows:

```
public void paintComponent(Graphics g)
```

The Graphics object g is created automatically by the Java runtime system. This object controls how information is drawn. You can use various drawing methods defined in the Graphics class to draw strings and geometric figures. For example, you can draw a string using the following method in the Graphics class:

```
drawString(string, x1, y1);
```

NOTE: The Graphics class is an abstract class for displaying figures and images on the screen in different platforms. The Graphics class is implemented in the native platform in JVM. When you use the paintComponent method to draw things on a graphics context g, this g is an instance of a concrete subclass of the abstract Graphics class for the specific platform. The Graphics class encapsulates the platform details and enables you to draw things uniformly without concerning specific platforms.

All the drawing methods have arguments that specify the locations of the subjects to be drawn. The Java coordinate system has x in the horizontal axis and y in the vertical

axis, with the origin (0, 0) at the upper-left corner of the window. The x coordinate increases to the right, and the y coordinate increases downward. All measurements in Java are made in pixels, as shown in Figure 10.11.

*****Insert Figure 10.11 (Same as Figure 8.11 in the 3rd Edition, p303)**

Figure 10.11

The Java graphics coordinate system is measured in pixels, with (0, 0) at its upper-left corner.

You can draw things using appropriate colors and fonts. The next sections introduce the Color class, the Font class, the FontMetrics class, and drawing methods in the Graphics class.

The Color Class

You can set colors for GUI components by using the java.awt.Color class. Colors are made of red, green, and blue components, each of which is represented by a byte value that describes its intensity, ranging from 0 (darkest shade) to 255 (lightest shade). This is commonly known as the RGB model.

The syntax to create a Color object is

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

in which r, g, and b specify a color by its red, green, and blue components; for example:

```
Color color = new Color(128, 100, 100);
```

You can use the setBackground(Color c) and setForeground(Color c) methods defined in the Component class to set a component's background and foreground colors. Following is an example of setting the background of a panel by using Color c:

```
JPanel myPanel = new JPanel();  
myPanel.setBackground(c);
```

Alternatively, you can use one of the 13 standard colors (black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow) defined as constants in java.awt.Color. For example, the following code sets the background color of a panel to yellow.

```
JPanel myPanel = new JPanel();  
myPanel.setBackground(Color.yellow);
```

NOTE: The standard color names are constants, but they are named as variables with lowercase for the first word and uppercase for the first letters of subsequent words. Thus the color names violate the Java naming convention.

The Font and FontMetrics Classes

You can set fonts for the components, or subjects you draw, and use font metrics to measure font size. Fonts and font metrics are encapsulated in the classes Font and FontMetrics.

Whatever font is current will be used in the subsequent drawing. To set a font, you need to create a Font object from the Font class. The syntax is:

```
Font myFont = new Font(name, style, size);
```

You can choose a font name from SansSerif, Serif, Monospaced, Dialog, or DialogInput, and choose a style from Font.PLAIN, Font.BOLD, and Font.ITALIC. The styles can be combined, as in the following code:

```
Font myFont = new Font("SansSerif", Font.BOLD, 16);  
Font myFont = new Font("Serif", Font.BOLD+Font.ITALIC, 12);
```

FontMetrics can be used to compute the exact length and width of a string, which is helpful for measuring the size of a string in order to display it in the right position. For example, you can center strings in the viewing area with the help of the FontMetrics class. A FontMetrics is measured by the following attributes (see Figure 10.12):

- **Leading:** Pronounced *ledding*, this is the amount of space between lines of text.
- **Ascent:** This is the height of a character, from the baseline to the top.
- **Descent:** This is the distance from the baseline to the bottom of a descending character, such as *j*, *y*, and *g*.
- **Height:** This is the sum of leading, ascent, and descent.

***Insert Figure 10.12 (Same as Figure 8.12 in the 3rd Edition, p304)

Figure 10.12

The FontMetrics class can be used to determine the font properties of characters.

FontMetrics is an abstract class. To get a FontMetrics object for a specific font, use the following getFontMetrics methods defined in the Graphics class.

- `public FontMetrics getFontMetrics(Font f)`

This method returns the font metrics of the specified font.

- `public FontMetrics getFontMetrics()`

This method returns the font metrics of the current font.

You can use the following instance methods in the FontMetrics class to obtain font information:

```
public int getAscent()  
public int getDescent()  
public int getLeading()  
public int getHeight()  
public int stringWidth(String str)
```

Example 10.5 Using FontMetrics

This example presents a program that displays "Welcome to Java" in SansSerif 20-point bold, centered in the frame. The output of the program is shown in Figure 10.13.

```
// TestFontMetrics.java: Draw a message at the center of a panel  
package chapter10;  
  
import java.awt.Font;  
import java.awt.FontMetrics;  
import java.awt.Graphics;  
import javax.swing.*;  
  
public class TestFontMetrics extends JFrame  
{  
    /**Default constructor*/  
    public TestFontMetrics()  
    {  
        MessagePanel messagePanel = new MessagePanel("Welcome to Java");  
  
        // Set font SansSerif 20-point bold  
        messagePanel.setFont(new Font("SansSerif", Font.BOLD, 20));  
  
        // Center the message  
        messagePanel.setCentered(true);  
  
        getContentPane().add(messagePanel);  
    }  
  
    /**Main method*/  
    public static void main(String[] args)  
    {  
        TestFontMetrics frame = new TestFontMetrics();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(300, 200);  
        frame.setTitle("TestFontMetrics");  
        frame.setVisible(true);  
    }  
}
```

*****Layout: Insert a separator line here. AU**

```
// MessagePanel.java: Display a message on a JPanel
package chapter10;

import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JPanel;

public class MessagePanel extends JPanel
{
    /**The message to be displayed*/
    private String message = "Welcome to Java";

    /**The x coordinate where the message is displayed*/
    private int xCoordinate = 20;

    /**The y coordinate where the message is displayed*/
    private int yCoordinate = 20;

    /**Indicate whether the message is displayed in the center*/
    private boolean centered;

    /**Default constructor*/
    public MessagePanel()
    {
    }

    /**Constructor with a message parameter*/
    public MessagePanel(String message)
    {
        this.message = message;
    }

    /**Return message*/
    public String getMessage()
    {
        return message;
    }

    /**Set a new message*/
    public void setMessage(String message)
    {
        this.message = message;
    }

    /**Return xCoordinator*/
    public int getXCoordinate()
    {
        return xCoordinate;
    }

    /**Set a new xCoordinator*/
    public void setXCoordinate(int x)
    {
        this.xCoordinate = x;
    }

    /**Return yCoordinator*/
    public int getYCoordinate()
    {
        return yCoordinate;
    }

    /**Set a new yCoordinator*/
    public void setYCoordinate(int y)
    {
        this.yCoordinate = y;
    }

    /**Return centered*/
    public boolean isCentered()
    {
    }
}
```

```

    return centered;
}

/**Set a new centered*/
public void setCentered(boolean centered)
{
    this.centered = centered;
}

/**Paint the message*/
public void paintComponent(Graphics g)
{
    super.paintComponent(g);

    if (centered)
    {
        // Get font metrics for the current font
        FontMetrics fm = g.getFontMetrics();

        // Find the center location to display
        int w = fm.stringWidth(message); // Get the string width
        // Get the string height, from top to the baseline
        int h = fm.getAscent();
        xCoordinate = getWidth()/2-w/2;
        yCoordinate = getHeight()/2+h/2;
    }

    g.drawString(message, xCoordinate, yCoordinate);
}

/**Override get method for preferredSize*/
public Dimension getPreferredSize()
{
    return new Dimension(200, 100);
}

/**Override get method for minimumSize*/
public Dimension getMinimumSize()
{
    return new Dimension(200, 100);
}
}

```

*****Insert Figure 10.13 (Same as Figure 8.13 in the 3rd Edition, 307)**

Figure 10.13

The program uses the FontMetrics class to measure the string width and height, and displays it at the center of the frame.

Example Review

The example contains two classes: TestFontMetrics and MessagePanel. The relationship between the two classes is shown in Figure 10.14.

*****Insert Figure 10.14**

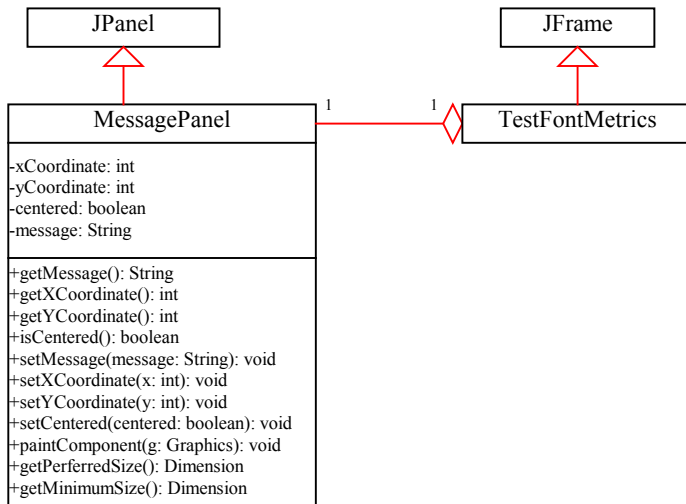


Figure 10.14

TestFontMetrics uses MessagePanel to display a message.

TestFontMetrics creates an instance of MessagePanel to display a message at the center of the panel. The setFont method sets a new font for the message panel. This method is available for all subclasses of Component. The new font is created using new Font("SansSerif", Font.BOLD, 20). This new font is used to display the message in the panel.

The MessagePanel class has the properties message, xCoordinate, yCoordinate, and centered. xCoordinate and yCoordinate specify where the message is displayed if centered is false. If centered is true, the message is displayed at the center of the panel.

The methods getWidth() and getHeight(), defined in the Component class, return this component's width and height, respectively.

Since the centered property is set to true in TestFontMetrics, the message is displayed in the center of the panel. Resizing the frame results in the message always being displayed in the center of the panel.

yCoordinate is height of the baseline for the first character of the string to be displayed. When centered is true, yCoordinate should be getHeight()/2 + h/2, where h is the ascent of the string.

The getPreferredSize() method, defined in Component, is overridden in MessagePanel to specify a preferred

size for the layout manager to consider when laying out a MessagePanel object.

The drawString method draws on the panel. The drawString(s, x, y) method draws a string s whose left end of the baseline starts at (x, y).

The Swing components use the paintComponent method to draw things. The paintComponent method is invoked to paint the graphics context. This method is invoked when the frame is displayed or whenever you resize the frame. Invoking super.paintComponent(g) is necessary to clear the viewing area before a new drawing is displayed. If this method is not invoked, the previous drawings will not be cleared.

There are many ways to write Java programs. You can rewrite this example using one class. See Exercise 10.17.

CAUTION: The MessagePanel class uses the properties xCoordinate and yCoordinate to specify the position of the message displayed on the panel. Do not use property names x, and y, because they are already defined in the Component class to specify the position of the component in the parent's coordinate system.

NOTE: The Component class has the setBackground, setForeground and setFont methods. These methods are for setting colors and fonts for the entire component. Suppose you want to draw several messages in a panel with different colors and fonts, you have to use the setColor and setFont methods in the Graphics class to set the color and font for the current drawing.

NOTE: One of the key features of Java programming is reusing classes. Throughout the book, you will develop reusable classes and reuse them later. MessagePanel is one such example. It can be reused whenever you need to display a message on a panel.

Drawing Geometric Figures

This section introduces the methods in the Graphics class for drawing lines, rectangles, ovals, arcs, and polygons.

Drawing Lines

You can use the method shown below to draw a straight line:

```
drawLine(x1, y1, x2, y2);
```

The parameters x1, y1, x2, and y2 represent the starting point (x1, y1) and the ending point (x2, y2) of the line, as shown in Figure 10.15.

*****Insert Figure 10.15 (Same as Figure 8.15 in the 3rd Edition, p309)**

Figure 10.15

The drawLine method draws a line between two specified points.

Drawing Rectangles

Java provides six methods for drawing rectangles in outline or filled with color. You can draw plain rectangles, rounded rectangles, or 3D rectangles.

To draw a plain rectangle, use:

```
drawRect(x, y, w, h);
```

To draw a rectangle filled with color, use the following code:

```
fillRect(x, y, w, h);
```

The parameters x, and y represent the upper-left corner of the rectangle, and w and h are the width and height of the rectangle (see Figure 10.16).

*****Insert Figure 10.16 (Same as Figure 8.16 in the 3rd Edition, p310)**

Figure 10.16

The drawRect method draws a rectangle with specified upper-left corner (x, y), width, and height.

To draw a rounded rectangle, use the following method:

```
drawRoundRect(x, y, w, h, aw, ah);
```

To draw a rounded rectangle filled with color, use this code:

```
fillRoundRect(x, y, w, h, aw, ah);
```

Parameters x, y, w, and h are the same as in the drawRect method, parameter aw is the horizontal diameter of the arcs at the corner, and ah is the vertical diameter of the arcs at the corner (see Figure 10.17).

*****Insert Figure 10.17 (Same as Figure 8.17 in the 3rd Edition, p311)**

Figure 10.17

The drawRoundRect method draws a rounded-corner rectangle.

To draw a 3D rectangle, use

```
draw3DRect(x, y, w, h, raised);
```

in which x, y, w, and h are the same as in the drawRect method. The last parameter, a Boolean value, indicates whether the rectangle is raised or indented from the surface.

The example given below demonstrates these methods. The output is shown in Figure 10.18.

```
// TestRect.java: Demonstrate drawing rectangles  
package chapter10;  
  
import java.awt.Graphics;  
import java.awt.Color;  
import javax.swing.JPanel;  
import javax.swing.JFrame;  
  
public class TestRect extends JFrame  
{  
    /**Default constructor*/  
    public TestRect()  
    {  
        setTitle("Show Rectangles");  
        getContentPane().add(new RectPanel());  
    }  
  
    /**Main method*/  
    public static void main(String[] args)  
    {  
        TestRect frame = new TestRect();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(300,250);  
        frame.setVisible(true);  
    }  
}  
  
class RectPanel extends JPanel  
{  
    public void paintComponent(Graphics g)  
    {  
        super.paintComponent(g);  
  
        // Set new color  
        g.setColor(Color.red);  
  
        // Draw a rectangle  
        g.drawRect(5, 5, getWidth()/2-10, getHeight()/2-10);  
  
        // Draw a rounded rectangle  
        g.drawRoundRect(getWidth()/2+5, 5,  
            getWidth()/2-10, getHeight()/2-10, 60, 30);  
  
        // Change the color to cyan  
        g.setColor(Color.cyan);  
  
        // Draw a 3D rectangle  
        g.fill3DRect(5, getHeight()/2+5, getWidth()/2-10,  
            getHeight()/2-10, true);  
  
        // Draw a raised 3D rectangle
```

```

        g.fill3DRect(getWidth()/2+5, getHeight()/2+5, getWidth()/2-10,
                     getHeight()/2-10, false);
    }
}

```

*****Insert Figure 10.18**

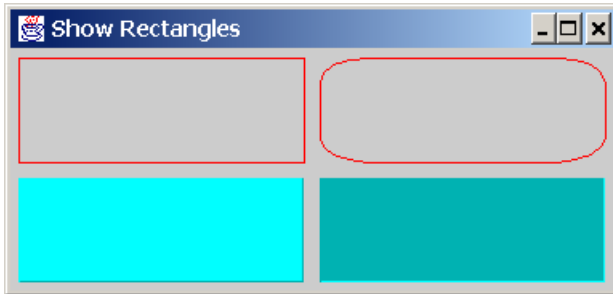


Figure 10.18

The program draws a rectangle, a rounded rectangle, and a 3D rectangle.

NOTE: The RectPanel class will be reused in Exercises 10.15, and 10.16.

Drawing Ovals

You can use the methods drawOval or fillOval to draw an oval in outline or filled solid. In Java, the oval is drawn based on its bounding rectangle; therefore, give the parameters as if you were drawing a rectangle.

Here is the method for drawing an oval:

```
drawOval(x, y, w, h);
```

To draw a filled oval, use the following method:

```
fillOval(x, y, w, h);
```

Parameters x and y indicate the top-left corner of the bounding rectangle, and w and h indicate the width and height, respectively, of the bounding rectangle, as shown in Figure 10.19.

*****Insert Figure 10.19 (Same as Figure 8.19 in the 3rd Edition, p312)**

Figure 10.19

The drawOval method draws an oval based on its bounding rectangle.

The following is an example of how to draw ovals, with the output in Figure 10.20.

```
// TestOvals.java: Demonstrate drawing ovals
package chapter10;

import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Graphics;

public class TestOvals extends JFrame
{
    /**Default constructor*/
    public TestOvals()
    {
        setTitle("Show Ovals");
        getContentPane().add(new OvalsPanel());
    }

    /**Main method*/
    public static void main(String[] args)
    {
        TestOvals frame = new TestOvals();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(250, 250);
        frame.setVisible(true);
    }
}

// The class for drawing the ovals on a panel
class OvalsPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        g.drawOval(5, 5, getWidth()/2-10, getHeight()/2-10);
        g.setColor(Color.red);
        g.drawOval(getWidth()/2+5, 5, getWidth()/2-10, getHeight()/2-10);
        g.setColor(Color.yellow);
        g.fillOval(5, getHeight()/2+5, getWidth()/2-10,
            getHeight()/2-10);
        g.setColor(Color.orange);
        g.fillOval(getWidth()/2+5, getHeight()/2+5, getWidth()/2-10,
            getHeight()/2-10);
    }
}
```

*****Insert Figure 10.20**

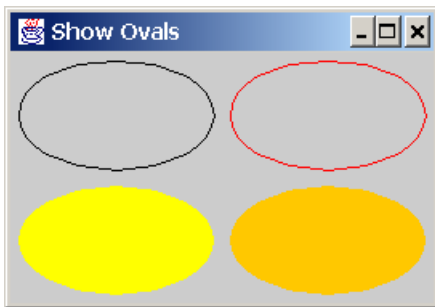


Figure 10.20

The program draws an oval, a circle, and a filled oval.

Drawing Arcs

Like an oval, an arc is drawn based on its bounding rectangle. An arc is conceived as part of an oval. The methods to draw or fill an arc are as follows:

```
drawArc(x, y, w, h, angle1, angle2);
```

```
fillArc(x, y, w, h, angle1, angle2);
```

Parameters x, y, w, and h are the same as in the drawOval method; parameter angle1 is the starting angle; angle2 is the spanning angle (that is, the angle covered by the arc). Angles are measured in degrees and follow the usual mathematical conventions (that is, 0 degrees is at 3 o'clock, and positive angles indicate counterclockwise rotation; see Figure 10.21).

*****Insert Figure 10.21 (Same as Figure 8.21 in the 3rd Edition, p314)**

Figure 10.21

The drawArc method draws an arc based on an oval with specified angles.

Shown below is an example of how to draw arcs; the output is shown in Figure 10.22.

```
// TestArcs.java: Demonstrate drawing arcs  
package chapter10;  
  
import javax.swing.JFrame;  
import javax.swing.JPanel;  
import java.awt.Color;  
import java.awt.Graphics;  
  
public class TestArcs extends JFrame  
{  
    /**Default constructor*/  
    public TestArcs()  
    {  
        setTitle("Show Arcs");  
        getContentPane().add(new ArcsPanel());  
    }  
  
    /**Main method*/  
    public static void main(String[] args)  
    {  
        TestArcs frame = new TestArcs();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(250, 300);  
        frame.setVisible(true);  
    }  
}  
  
// The class for drawing arcs on a panel  
class ArcsPanel extends JPanel  
{  
    // Draw four blades of a fan  
    public void paintComponent(Graphics g)  
    {  
        super.paintComponent(g);  
  
        int xCenter = getWidth()/2;  
        int yCenter = getHeight()/2;
```

```

    int radius =
        (int)(Math.min(getWidth(), getHeight())*0.4);

    int x = xCenter - radius;
    int y = yCenter - radius;

    g.fillArc(x, y, 2*radius, 2*radius, 0, 30);
    g.fillArc(x, y, 2*radius, 2*radius, 90, 30);
    g.fillArc(x, y, 2*radius, 2*radius, 180, 30);
    g.fillArc(x, y, 2*radius, 2*radius, 270, 30);
}

```

*****Insert Figure 10.22 (Same as Figure 8.22 in the 3rd Edition, p315)**

Figure 10.22

The program draws four filled arcs.

Drawing Polygons

The Polygon class encapsulates a description of a closed, two-dimensional region within a coordinate space. This region is bounded by an arbitrary number of line segments, each of which is one side (or edge) of the polygon. Internally, a polygon comprises a list of (x, y) coordinate pairs in which each pair defines a vertex of the polygon, and two successive pairs are the endpoints of a line that is a side of the polygon. The first and final pairs of (x, y) points are joined by a line segment that closes the polygon. Java enables you to draw a polygon in two ways: by using the direct method or by using the Polygon object.

The direct method draws a polygon by specifying all the points, using the following two methods:

```

drawPolygon(x, y, n);

fillPolygon(x, y, n);

```

Parameters x and y are arrays representing x-coordinates and y-coordinates, and n indicates the number of points. For example:

```

int x[] = {40, 70, 60, 45, 20};
int y[] = {20, 40, 80, 45, 60};
g.drawPolygon(x, y, x.length);
g.fillPolygon(x, y, x.length);

```

The drawing method opens the polygon by drawing lines between point (x[i], y[i]) and point (x[i+1], y[i+1]) for i = 0, ..., length-1; it closes the polygon by drawing a line between the first and last points (see Figure 10.23).

*****Insert Figure 10.23 (Same as Figure 8.23 in the 3rd Edition, p316)**

Figure 10.23

The drawPolygon method draws a polygon with specified points.

You can also draw a polygon by first creating a Polygon object, then adding points to it, and finally displaying it. To create a Polygon object, use

```
Polygon poly = new Polygon();
```

Or

```
Polygon poly = new Polygon(x, y, n);
```

Parameters x, y, and n are the same as in the previous drawPolygon method. Here is an example of how to draw a polygon in Graphics g:

```
Polygon polygon = new Polygon();  
polygon.addPoint(20, 30);  
polygon.addPoint(40, 40);  
polygon.addPoint(50, 50);  
g.drawPolygon(polygon);
```

The addPoint method adds a point to the polygon. The drawPolygon method also takes a Polygon object as a parameter.

Next is an example of how to draw a polygon with the output shown in Figure 10.24.

```
// TestPolygon.java: Demonstrate drawing polygons  
package chapter10;  
  
import javax.swing.JFrame;  
import javax.swing.JPanel;  
import java.awt.Graphics;  
import java.awt.Polygon;  
  
public class TestPolygon extends JFrame  
{  
    /**Default constructor*/  
    public TestPolygon()  
    {  
        setTitle("Show Polygon");  
        getContentPane().add(new PolygonsPanel());  
    }  
  
    /**Main method*/  
    public static void main(String[] args)  
    {  
        TestPolygon frame = new TestPolygon();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(200, 250);  
        frame.setVisible(true);  
    }  
}  
  
// Draw a polygon in the panel  
class PolygonsPanel extends JPanel  
{  
    public void paintComponent(Graphics g)  
    {  
        super.paintComponent(g);  
  
        int xCenter = getWidth()/2;
```

```

    int yCenter = getHeight()/2;
    int radius =
        (int)(Math.min(getWidth(), getHeight())*0.4);

    // Create a Polygon object
    Polygon polygon = new Polygon();

    // Add points to the polygon
    polygon.addPoint(xCenter + radius, yCenter);
    polygon.addPoint((int)(xCenter + radius*Math.cos(2*Math.PI/6)),
        (int)(yCenter - radius*Math.sin(2*Math.PI/6)));
    polygon.addPoint((int)(xCenter + radius*Math.cos(2*2*Math.PI/6)),
        (int)(yCenter - radius*Math.sin(2*2*Math.PI/6)));
    polygon.addPoint((int)(xCenter + radius*Math.cos(3*2*Math.PI/6)),
        (int)(yCenter - radius*Math.sin(3*2*Math.PI/6)));
    polygon.addPoint((int)(xCenter + radius*Math.cos(4*2*Math.PI/6)),
        (int)(yCenter - radius*Math.sin(4*2*Math.PI/6)));
    polygon.addPoint((int)(xCenter + radius*Math.cos(5*2*Math.PI/6)),
        (int)(yCenter - radius*Math.sin(5*2*Math.PI/6)));

    // Draw the polygon
    g.drawPolygon(polygon);
}

```

*****Insert Figure 10.24 (Same as Figure 8.24 in the 3rd Edition, p317)**

Figure 10.24

The program uses the drawPolygon method to draw a polygon.

NOTE: Prior to JDK 1.1, a polygon was a sequence of lines that were not necessarily closed. But in JDK 1.1, a polygon is always closed. Nevertheless, you can draw a nonclosed polygon by using the drawPolyline(int[] x, int[] y, int nPoints) method, which draws a sequence of connected lines defined by arrays of x and y coordinates. The figure is not closed if the first point differs from the last point.

Case Studies

This case study presents an example that uses several drawing methods and trigonometric methods to draw a clock showing the current time in a frame. To draw a clock, you need to draw a circle and three hands for second, minute, and hour. To draw a hand, you need to specify the two ends of the line. As shown in Figure 10.25, one end is the center of the clock at (xCenter, yCenter), and the other end at (xEnd, yEnd) is determined by the following formula:

```

xEnd = xCenter + handLength × sin(π)
yEnd = yCenter - handLength × cos(π)

```

*****Insert Figure 10.25 (Same as Figure 8.25 in the 3rd Edition, p318)**

Figure 10.25

The end point of a clock hand can be determined given the spanning angle, the hand length, and the center point.

Angle θ is in radians. Let second, minute, and hour denote the current second, minute, and hour.

The angle for the second hand is

$$\text{second} \times (2\pi/60)$$

The angle for the minute hand is

$$(\text{minute} + \text{second}/60) \times (2\pi/60)$$

The angle for hour hand is

$$(\text{hour} + \text{minute}/60 + \text{second}/(60[ts]60)) \times (2\pi/12)$$

For simplicity, you can omit the seconds in computing the angles of the minute hand and the hour hand, since they are very small and can be neglected. Therefore the end points for the second hand, minute hand, and hour hand can be computed as:

```
xSecond = xCenter + secondHandLength × sin(second × (2π/60))
ySecond = yCenter - secondHandLength × cos(second × (2π/60))
xMinute = xCenter + minuteHandLength × sin(minute × (2π/60))
yMinute = yCenter - minuteHandLength × cos(minute × (2π/60))
xHour = xCenter + hourHandLength × sin((hour + minute/60) (2π/60))
yHour = yCenter - hourHandLength × cos((hour + minute/60) × (2π/60))
```

Example 10.6 Drawing a Clock

This example presents a program that displays a clock based on the specified hour, minute, and second. The hour, minute, and second are passed to the program as command-line arguments like this:

```
java DisplayClock hour minute second
```

The program is given next, and its output is shown in Figure 10.26.

```
// DisplayClock.java: Display a clock in a panel
package chapter10;

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

public class DisplayClock extends JFrame
{
    /**Main method to display hour, minute and hour
     * @param args[0] hour
     * @param args[1] minute
     * @param args[2] second
     */
    public static void main(String[] args)
    {
        // Declare hour, minute, and second values
        int hour = 0;
        int minute = 0;
        int second = 0;

        // Check usage and get hour, minute, second
        if (args.length > 3)
```

```

    {
        System.out.println(
            "Usage: java DisplayClock hour minute second");
        System.exit(0);
    }
    else if (args.length == 3)
    {
        hour = new Integer(args[0]).intValue();
        minute = new Integer(args[1]).intValue();
        second = new Integer(args[2]).intValue();
    }
    else if (args.length == 2)
    {
        hour = new Integer(args[0]).intValue();
        minute = new Integer(args[1]).intValue();
    }
    else if (args.length == 1)
    {
        hour = new Integer(args[0]).intValue();
    }
    }

    // Create a frame to hold the clock
    DisplayClock frame = new DisplayClock();
    frame.setTitle("Display Clock");
    frame.getContentPane().add(new DrawClock(hour, minute, second));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 350);
    frame.setVisible(true);
}
}

```

*****Layout: Insert a separator line here. AU**

```

// DrawClock.java: Display a clock in JPanel
package chapter10;

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

public class DrawClock extends JPanel
{
    private int hour;
    private int minute;
    private int second;
    protected int xCenter, yCenter;
    protected int clockRadius;

    /**Construct a clock panel*/
    public DrawClock(int hour, int minute, int second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    /**Draw the clock*/
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        // Initialize clock parameters
        clockRadius =
            (int) (Math.min(getWidth(), getHeight())*0.8*0.5);
        xCenter = (getWidth())/2;
        yCenter = (getHeight())/2;

        // Draw circle
        g.setColor(Color.black);
        g.drawOval(xCenter - clockRadius, yCenter - clockRadius,
            2*clockRadius, 2*clockRadius);
        g.drawString("12", xCenter-5, yCenter-clockRadius+12);
        g.drawString("9", xCenter-clockRadius+3, yCenter+5);
        g.drawString("3", xCenter+clockRadius-10, yCenter+3);
        g.drawString("6", xCenter-3, yCenter+clockRadius-3);

        // Draw second hand
    }
}

```

```

        int sLength = (int)(clockRadius*0.8);
        int xSecond =
            (int)(xCenter + sLength*Math.sin(second*(2*Math.PI/60)));
        int ySecond =
            (int)(yCenter - sLength*Math.cos(second*(2*Math.PI/60)));
        g.setColor(Color.red);
        g.drawLine(xCenter, yCenter, xSecond, ySecond);

        // Draw minute hand
        int mLength = (int)(clockRadius*0.65);
        int xMinute =
            (int)(xCenter + mLength*Math.sin(minute*(2*Math.PI/60)));
        int yMinute =
            (int)(yCenter - mLength*Math.cos(minute*(2*Math.PI/60)));
        g.setColor(Color.blue);
        g.drawLine(xCenter, yCenter, xMinute, yMinute);

        // Draw hour hand
        int hLength = (int)(clockRadius*0.5);
        int xHour = (int)(xCenter +
            hLength*Math.sin((hour+minute/60.0)*(2*Math.PI/12)));
        int yHour = (int)(yCenter -
            hLength*Math.cos((hour+minute/60.0)*(2*Math.PI/12)));
        g.setColor(Color.green);
        g.drawLine(xCenter, yCenter, xHour, yHour);

        // Display current time in string
        g.setColor(Color.red);
        String time = "Hour: " + hour + " Minute: " + minute +
            " Second: " + second;
        FontMetrics fm = g.getFontMetrics();
        g.drawString(time, (getWidth() -
            fm.stringWidth(time))/2, yCenter+clockRadius+30);
    }
}

```

*****Insert Figure 10.26**

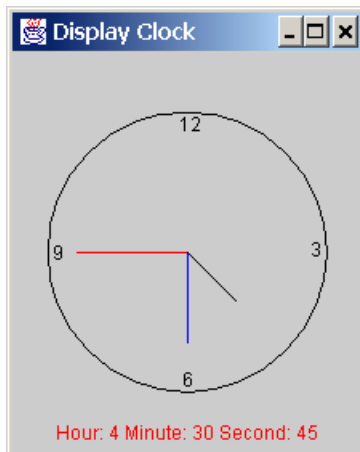


Figure 10.26

The program displays a clock that shows the time with the specified hour, minute, and second.

Example Review

The DisplayClock class obtains command-line arguments for hour, minute, and second, and uses this information to create an instance of DrawClock.

DrawClock is responsible for drawing the clock in a panel.

The program enables the clock size to adjust as the frame resizes. Every time you resize the frame, the paintComponent method is automatically invoked to paint the new frame. The paintComponent method displays the clock in proportion to the frame size.

The numeric time (consisting of hour, minute, and second) is displayed below the clock. The program uses font metrics to determine the size of the time string and display it in the center.

Event-Driven Programming

All the programs you have written so far are object-oriented, but executed in a procedural order. You used decision and loop statements to control the flow of execution, but the program dictated the flow of execution. Java graphics programming is event-driven. In event-driven programming, code is executed when an event—a button click, perhaps, or a mouse movement—occurs. This section introduces the Java event model.

Event and Event Source

When you run Java graphics programs, the program interacts with the user and the events drive the execution of the program. An event can be defined as a signal to the program that something has happened. The event is generated either by external user actions, such as mouse movements, mouse button clicks, and keystrokes, or by the operating system, such as a timer. The program can choose to respond to or ignore the event.

The GUI component on which the event is generated is called the *source object*. For example, a button is the source object for the clicking-button action event. An event is an instance of an event class. The root class of the event classes is java.util.EventObject. The hierarchical relationships of the event classes used in this book are shown in Figure 10.27.

*****Insert Figure 10.27 (Same as Figure 8.27 in the 3rd Edition, p323)**

Figure 10.27

An event is an object of the EventObject class.

An event object contains whatever properties that are pertinent to the event. You can identify the source object of the event using the `getSource()` instance method in the `EventObject` class. The subclasses of `EventObject` deal with special types of events such as button actions, window events, component events, mouse movements, and keystrokes. Table 10.1 lists external user actions, source objects, and event types generated.

***Layout: Same as jbBook. AU ***
--

Table 10.1 User Action, Source Object, and Event Type

User Action	Source Object	Event Type Generated
Click a button	<u>JButton</u>	<u>ActionEvent</u>
Change text	<u>JTextComponent</u>	<u>TextEvent</u>
Press return on a text field	<u>TextField</u>	<u>ActionEvent</u>
Select a new item	<u>JComboBox</u>	<u>ItemEvent</u> , <u>ActionEvent</u>
Select item(s)	<u>JList</u>	<u>ListSelectionEvent</u>
Click a check box	<u>JCheckBox</u>	<u>ItemEvent</u> , <u>ActionEvent</u>
Click a radio button	<u>JRadioButton</u>	<u>ItemEvent</u> , <u>ActionEvent</u>
Select a menu item	<u>JMenuItem</u>	<u>ActionEvent</u>
Move the scroll bar	<u>JScrollBar</u>	<u>AdjustmentEvent</u>
Window opened, closed, iconified, deiconified, or closing	<u>Window</u>	<u>WindowEvent</u>
Component added or removed from the container	<u>Container</u>	<u>ContainerEvent</u>
Component moved, resized, hidden, or shown	<u>Component</u>	<u>ComponentEvent</u>
Component gained or lost focus	<u>Component</u>	<u>FocusEvent</u>
Key released or pressed	<u>Component</u>	<u>KeyEvent</u>
Mouse pressed, released, clicked, entered, or exited	<u>Component</u>	<u>MouseEvent</u>

Mouse moved, or dragged Component

MouseEvent

NOTE: If a component can generate an event, any subclass of the component can generate the same type of event. For example, every GUI component can generate MouseEvent, KeyEvent, FocusEvent, and ComponentEvent, since Component is the superclass of all GUI components.

NOTE: All the event classes in Figure 10.27 are included in the java.awt.event package except the ListSelectionEvent, which is in the javax.swing.event package. The AWT events were originally designed for AWT components, but many Swing components fire them.

Event Registration, Listening, and Handling

Java uses a delegation-based model for event handling: An external user action on a source object triggers an event. An object interested in the event receives the event. Such an object is called a *listener*. Not all objects can receive events. To become a listener, an object must be registered as a listener by the source object. The source object maintains a list of listeners and notifies all the registered listeners by invoking the event-handling method, or *handler*, on the listener object to respond to the event, as shown in Figure 10.28.

*****Insert Figure 10.28 (Same as Figure 8.28 in the 3rd Edition, p324)**

Figure 10.28

An event is triggered by user actions on the source object; the source object generates the event object and invokes the handler of the listener object to process the event.

For example, if a JFrame object is interested in the external events on a JButton source object, it must register with the JButton object. The registration is done by invoking a method from the JButton object to declare that the JFrame object is a listener for the JButton object. When you click the button, the JButton object generates an ActionEvent and notifies the listener by invoking a method defined in the listener to handle the event.

NOTE: A source object and a listener object may be the same. A source object may have many listeners. It maintains a queue for all of them.

Registration methods are dependent on event type. For ActionEvent, the method is addActionListener. In general, the method is named addXListener for XEvent. To become a listener, the listener must implement the standard handler. The handler is defined in the corresponding event-listener interface. Java provides a listener interface for every type of graphics event. For example, the corresponding listener interface for ActionEvent is ActionListener; each listener for ActionEvent should implement the ActionListener interface. Table 10.2 lists event types, the corresponding listener interfaces, and the methods defined in the listener interfaces.

***Layout: Same as jBook. AU ***

Table 10.2 Events, Event Listeners, and Listener Methods

Event Class	Listener Interface	Listener Methods (Handlers)
<u>ActionEvent</u>	<u>ActionListener</u>	<u>actionPerformed(ActionEvent e)</u>
<u>ItemEvent</u>	<u>ItemListener</u>	<u>itemStateChanged(ItemEvent e)</u>
<u>WindowEvent</u>	<u>WindowListener</u>	<u>windowClosing(WindowEvent e)</u> <u>windowOpened(WindowEvent e)</u> <u>windowIconified(WindowEvent e)</u> <u>windowDeiconified(WindowEvent e)</u> <u>windowClosed(WindowEvent e)</u> <u>windowActivated(WindowEvent e)</u> <u>windowDeactivated(WindowEvent e)</u>
<u>ContainerEvent</u>	<u>ContainerListener</u>	<u>componentAdded(ContainerEvent e)</u> <u>componentRemoved(ContainerEvent e)</u>
<u>ComponentEvent</u>	<u>ComponentListener</u>	<u>componentMoved(ComponentEvent e)</u> <u>componentHidden(ComponentEvent e)</u> <u>componentResized(ComponentEvent e)</u> <u>componentShown(ComponentEvent e)</u>
<u>FocusEvent</u>	<u>FocusListener</u>	<u>focusGained(FocusEvent e)</u> <u>focusLost(FocusEvent e)</u>
<u>TextEvent</u>	<u>TextListener</u>	<u>textValueChanged(TextEvent e)</u>
<u>KeyEvent</u>	<u>KeyListener</u>	<u>keyPressed(KeyEvent e)</u> <u>keyReleased(KeyEvent e)</u> <u>keyTyped(KeyEvent e)</u>

<u>MouseEvent</u>	<u>MouseListener</u>	<u>mousePressed(MouseEvent e)</u> <u>mouseReleased(MouseEvent e)</u> <u>mouseEntered(MouseEvent e)</u> <u>mouseExited(MouseEvent e)</u> <u>mouseClicked(MouseEvent e)</u>
	<u>MouseMotionListener</u>	<u>mouseDragged(MouseEvent e)</u> <u>mouseMoved(MouseEvent e)</u>
<u>AdjustmentEvent</u>	<u>AdjustmentListener</u>	<u>adjustmentValueChanged</u> <u>(AdjustmentEvent e)</u>

NOTE: In general, the listener interface is named XListener for XEvent, except for MouseMotionListener.

Handling Events

A listener object must implement the corresponding listener interface. For example, a listener for a JButton source object must implement the ActionListener interface. The ActionListener interface contains the actionPerformed(ActionEvent e) method. This method must be implemented in the listener class. Upon receiving notification, it is executed to handle the event.

An event object is passed to the handling method. The event object contains information pertinent to the event type. You can get useful data values from the event object for processing the event. For example, you can use e.getSource() to obtain the source object in order to determine whether it is a button, a check box, a radio button, or a menu item.

Three examples of the use of event handling are given below. The first is for the ActionEvent, the second for the WindowEvent, and the third involves multiple listeners for a source.

Example 10.7 Handling Simple Action Events

This example presents a program that displays two buttons, OK and Cancel, in the window. A message is displayed on the console to indicate which button is clicked. Figure 10.29 shows a sample run of the program.

```
// TestActionEvent.java: Test ActionEvent
package chapter10;

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

```

public class TestActionEvent extends JFrame
    implements ActionListener
{
    // Create two buttons
    private JButton jbtOk = new JButton("OK");
    private JButton jbtCancel = new JButton("Cancel");

    /**Default constructor*/
    public TestActionEvent()
    {
        // Set the window title
        setTitle("TestActionEvent");

        // Set FlowLayout manager to arrange the components
        // inside the frame
        getContentPane().setLayout(new FlowLayout());

        // Add buttons to the frame
        getContentPane().add(jbtOk);
        getContentPane().add(jbtCancel);

        // Register listeners
        jbtOk.addActionListener(this);
        jbtCancel.addActionListener(this);
    }

    /**Main method*/
    public static void main(String[] args)
    {
        TestActionEvent frame = new TestActionEvent();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(100, 80);
        frame.setVisible(true);
    }

    /**This method will be invoked when a button is clicked*/
    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == jbtOk)
        {
            System.out.println("The OK button is clicked");
        }
        else if (e.getSource() == jbtCancel)
        {
            System.out.println("The Cancel button is clicked");
        }
    }
}

```

***Insert Figure 10.29

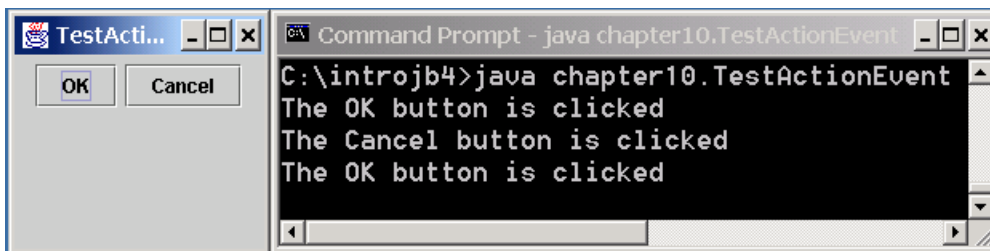


Figure 10.29

The program responds to the button action events.

Example Review

The button objects `jbtOk` and `jbtCancel` are the source of `ActionEvent`. The `TestActionEvent` class is a subclass of `JFrame` and is the listener for the action events. Therefore, `TestActionEvent` implements the `ActionListener` interface.

The statements

```
jbtOk.addActionListener(this);  
jbtCancel.addActionListener(this);
```

register `this` (referring to `TestActionEvent`) to listen to `ActionEvent` on `jbtOk` and `jbtCancel`.

Clicking a button causes the `actionPerformed` method to be invoked. The `e.getSource()` method is invoked to determine which button has been clicked.

The output in Figure 10.29 is displayed on the console when you run the program from a DOS prompt.

CAUTION: Missing listener registration is a common mistake in event handling. Because the source object doesn't notify the listener, the listener cannot act on the event.

TIP: To debug event-driven programs, you can insert a breakpoint at a statement in a handling method that you want to trace. For example, if you want to trace the `actionPerformed()` handler, insert a breakpoint at the first line in this method. When a button is clicked, the `actionPerformed()` handler is invoked, and the program pauses at the breakpoint, as shown in Figure 10.30.

*****Insert Figure 10.30**

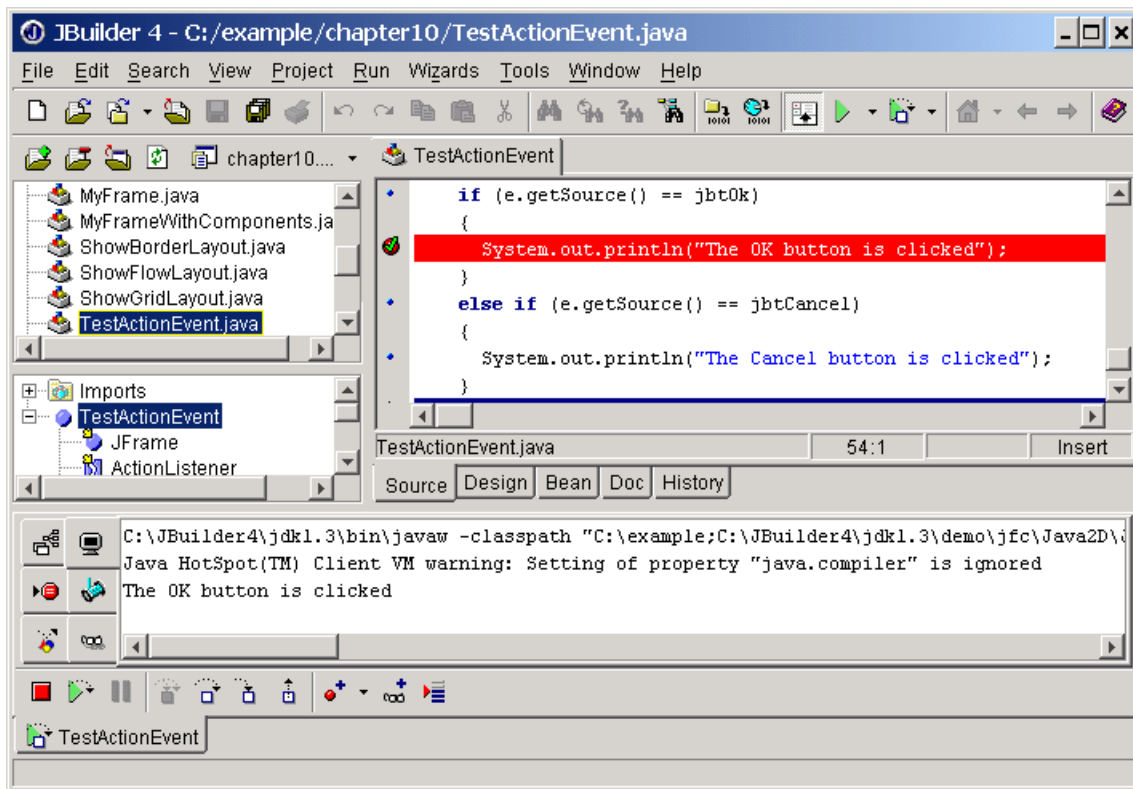


Figure 10.30

The program pauses at the breakpoint in the actionPerformed handler.

Example 10.8 Handling Window Events

This example demonstrates handling window events. Any subclass of the Window class can generate the following window events: window opened, closing, closed, activated, deactivated, iconified, and deiconified. This program creates a frame, listens to the window events, and displays a message to indicate the occurring event. Figure 10.31 shows a sample run of the program.

```

// TestWindowEvent.java: Create a frame to test window events
package chapter10;

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

public class TestWindowEvent extends JFrame
    implements WindowListener
{
    // Main method
    public static void main(String[] args)
    {
        TestWindowEvent frame = new TestWindowEvent();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("Test Window Event");
    }
}

```

```

    frame.setSize(100, 80);
    frame.setVisible(true);
}

// Default constructor
public TestWindowEvent()
{
    super();
    addWindowListener(this); // Register listener
}

/**
 * Handler for window deiconified event
 * Invoked when a window is changed from a minimized
 * to a normal state.
 */
public void windowDeiconified(WindowEvent event)
{
    System.out.println("Window deiconified");
}

/**
 * Handler for window iconified event
 * Invoked when a window is changed from a normal to a
 * minimized state. For many platforms, a minimized window
 * is displayed as the icon specified in the window's
 * iconImage property.
 * @see Frame#setIconImage
 */
public void windowIconified(WindowEvent event)
{
    System.out.println("Window iconified");
}

/**
 * Handler for window activated event
 * Invoked when the window is set to be the user's
 * active window, which means the window (or one of its
 * subcomponents) will receive keyboard events.
 */
public void windowActivated(WindowEvent event)
{
    System.out.println("Window activated");
}

/**
 * Handler for window deactivated event
 * Invoked when a window is no longer the user's active
 * window, which means that keyboard events will no longer
 * be delivered to the window or its subcomponents.
 */
public void windowDeactivated(WindowEvent event)
{
    System.out.println("Window deactivated");
}

/**
 * Handler for window opened event
 * Invoked the first time a window is made visible.
 */
public void windowOpened(WindowEvent event)
{
    System.out.println("Window opened");
}

/**
 * Handler for window closing event
 * Invoked when the user attempts to close the window
 * from the window's system menu. If the program does not
 * explicitly hide or dispose the window while processing
 * this event, the window close operation will be cancelled.
 */
public void windowClosing(WindowEvent event)
{
    System.out.println("Window closing");
}

/**
 * Handler for window closed event

```

```

    * Invoked when a window has been closed as the result
    * of calling dispose on the window.
    */
    public void windowClosed(WindowEvent event)
    {
        System.out.println("Window closed");
    }
}

```

*****Insert Figure 10.31 (Same as Figure 8.30 in the 3rd Edition, p330)**

Figure 10.31

The window events are displayed in the console when you run the program from a DOS prompt.

Example Review

The WindowEvent can be generated by the Window class or any subclasses of Window. Since JFrame is a subclass of Window, it can generate WindowEvent.

TestWindowEvent extends JFrame and implements WindowListener. The WindowListener interface defines several abstract methods (windowActivated, windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified, windowOpened) for handling window events when the window is activated, closed, closing, deactivated, deiconified, iconified, or opened.

When a window event, such as activation, occurs, the windowActivated method is triggered. You should implement the windowActivated method with a concrete response if you want this event to be processed.

Because all these methods in the WindowListener interface are abstract, you must implement all of them even if your program does not care about some of the events.

For an object to receive event notification, it must register as an event listener. addWindowListener(this) registers the object of TestWindowEvent as a window-event listener so that it can receive notification about the window event. TestWindowEvent is both a listener and a source object.

Example 10.9 Multiple Listeners for a Single Source

This example modifies Example 10.7 to add a new listener for each button. The two buttons OK and Cancel use the frame class as the listener. This

example creates a new listener class as an additional listener for the action events on the buttons. When a button is clicked, both listeners respond to the action event. Figure 10.32 shows a sample run of the program.

```
// TestMultipleListener.java: Test multiple listeners
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestMultipleListener extends JFrame
    implements ActionListener
{
    // Create two buttons
    private JButton jbtOk = new JButton("OK");
    private JButton jbtCancel = new JButton("Cancel");

    /**Default constructor*/
    public TestMultipleListener()
    {
        // Set the window title
        setTitle("TestMultipleListener");

        // Set FlowLayout manager to arrange the components
        // inside the frame
        getContentPane().setLayout(new FlowLayout());

        // Add buttons to the frame
        getContentPane().add(jbtOk);
        getContentPane().add(jbtCancel);

        // Register the frame as listeners
        jbtOk.addActionListener(this);
        jbtCancel.addActionListener(this);

        // Register a second listener for buttons
        SecondListener secondListener = new SecondListener();
        jbtOk.addActionListener(secondListener);
        jbtCancel.addActionListener(secondListener);
    }

    /**Main method*/
    public static void main(String[] args)
    {
        TestMultipleListener frame = new TestMultipleListener();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(100, 80);
        frame.setVisible(true);
    }

    /**This method will be invoked when a button is clicked*/
    public void actionPerformed(ActionEvent e)
    {
        System.out.print("First listener: ");

        if (e.getSource() == jbtOk)
        {
            System.out.println("The OK button is clicked");
        }
        else if (e.getSource() == jbtCancel)
        {
            System.out.println("The Cancel button is clicked");
        }
    }
}

/**The class for the second listener*/
class SecondListener implements ActionListener
{
    /**Handle ActionEvent*/
    public void actionPerformed(ActionEvent e)
    {
        System.out.print("Second listener: ");

        // A button has an actionCommand property, which is same as the
```



```

    // text of the button by default.
    if (e.getActionCommand().equals("OK"))
    {
        System.out.println("The OK button is clicked");
    }
    else if (e.getActionCommand().equals("Cancel"))
    {
        System.out.println("The Cancel button is clicked");
    }
}
}

```

***Insert Figure 10.32

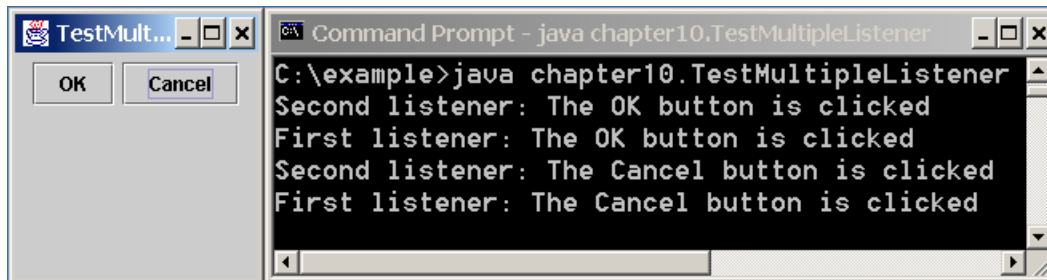


Figure 10.32

Both listeners respond to the button action events.

Example Review

Each source object in the previous two examples has a single listener. Each button in this example has two listeners: one is an instance of TestMultipleListener, and the other is an instance of SecondListener.

When a button is clicked, both listeners are notified and their respective actionPerformed methods are invoked. A button has an actionCommand property, whose default value is the label of the button. The getActionCommand method returns the button's actionCommand property. Using this method can detect which button is clicked. If you want to use the getSource method to detect which button is clicked, see Exercise 10.18.

The source object maintains a list to keep all its listeners. When a listener is registered with the source object, the listener is added to the top of the list. When an event occurs, the source object notifies the listener objects on the list by invoking each listener's handler. In this case, the handler is the actionPerformed method.

If you replace the highlighted code in this example with the following code:

```
// Register a second listener for buttons  
jbtOk.addActionListener(new SecondListener(this);  
jbtCancel.addActionListener(new SecondListener(this);
```

What would happen? Two instances of SecondListener are created. The program runs just as before the change.

Creating Java Applications Using the Application Wizard (Optional)

The Application wizard is often used to create graphics applications in JBuilder. The Application wizard creates an application consisting of two files and adds them to the current project. If no project is open, JBuilder runs the Project wizard first before it runs the Application wizard.

In this section, you will use the Application wizard to develop the same program for handling mouse events as in Example 10.7.

Here are the steps to complete the project:

1. With the AppBrowser for chapter10.jpr selected, choose File, New to display the Object Gallery, as shown in Figure 10.33.

*****Insert Figure 10.33**

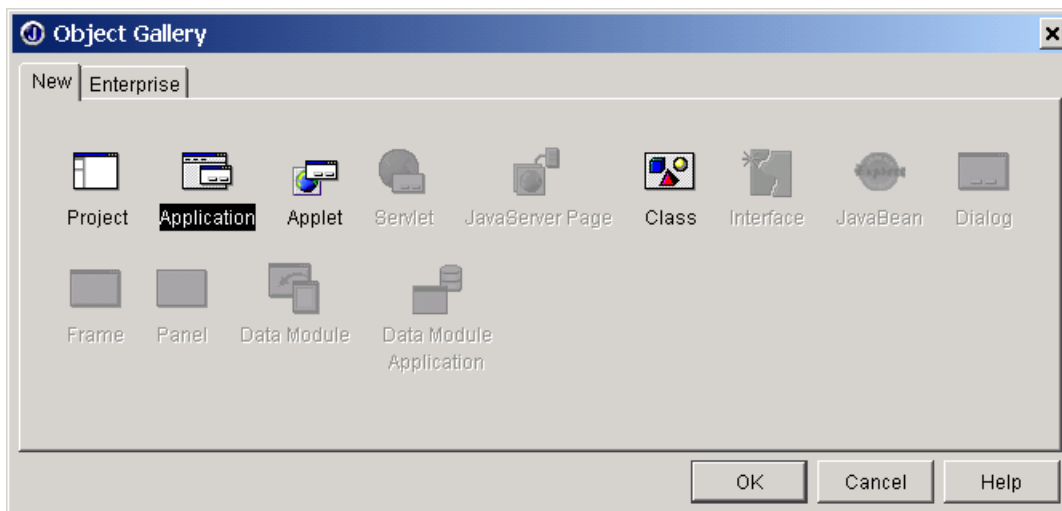


Figure 10.33

The Object Application wizard's Step 1 of 2 dialog box prompts you to enter a package name and an application main class name.

2. Click the Application icon to bring up the Application wizard, as shown in Figure 10.34.

*****Insert Figure 10.34**

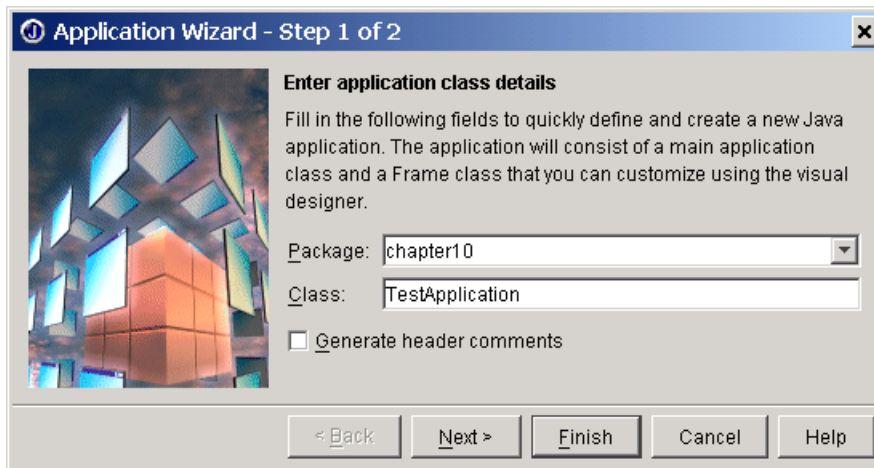


Figure 10.34

The Application wizard's Step 1 of 2 dialog box prompts you to enter a package name and an application main class name.

3. Type TestApplication in the Class field of the Application wizard's Step 1 of 2, check the option "Use only core JDK and Swing classes," and click Next to display the Application wizard's Step 2 of 2, as shown in Figure 10.35.

*****Insert Figure 10.35**

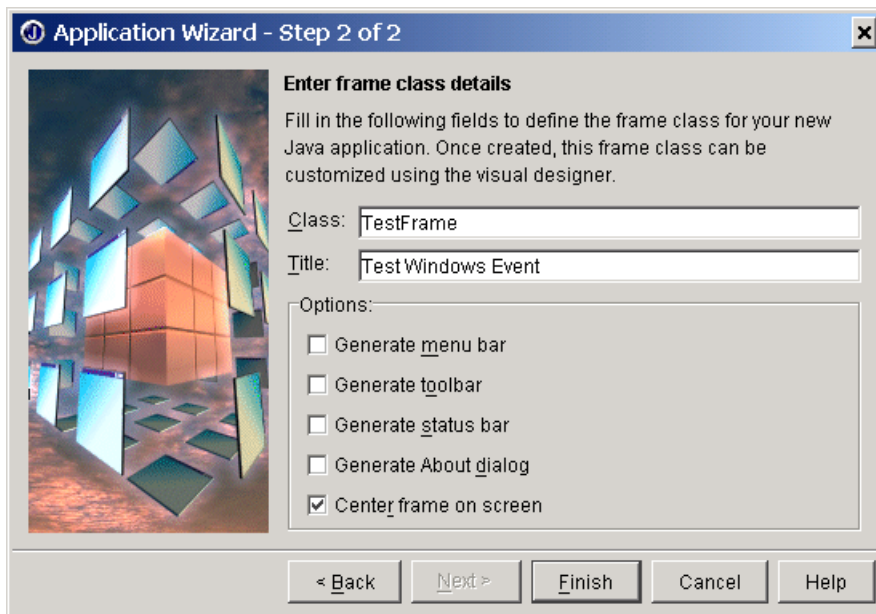


Figure 10.35

The Application wizard's Step 2 of 2 dialog box prompts you to enter the frame class name and specify frame style.

4. Type `TestFrame` in the Class name, check the option `Center frame on screen`, and click `Finish`.

The Application wizard created the following two files:

[BL] `TestApplication.java`, referred to as the *application class*.

[BL] `TestFrame.java`, referred to as the *frame class*.

These two files are shown as follows:

```
// TestApplication.java: Generated by the Application wizard
package chapter10;

import javax.swing.UIManager;
import java.awt.*;

public class TestApplication
{
    boolean packFrame = false;

    /**Construct the application*/
    public TestApplication()
    {
        TestFrame frame = new TestFrame();
        //Validate frames that have preset sizes
        //Pack frames that have useful preferred size info,
        //e.g. from their layout
        if (packFrame)
```

```

    {
        frame.pack();
    }
    else
    {
        frame.validate();
    }
    //Center the window
    Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = frame.getSize();
    if (frameSize.height > screenSize.height)
    {
        frameSize.height = screenSize.height;
    }
    if (frameSize.width > screenSize.width)
    {
        frameSize.width = screenSize.width;
    }
    frame.setLocation((screenSize.width - frameSize.width) / 2,
        (screenSize.height - frameSize.height) / 2);
    frame.setVisible(true);
}

/**Main method*/
public static void main(String[] args)
{
    try
    {
        UIManager.setLookAndFeel(
            UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    new TestApplication();
}

// TestFrame.java: Generated by the Application wizard
package chapter10;

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

public class TestFrame extends JFrame
{
    JPanel contentPane;
    BorderLayout borderLayout1 = new BorderLayout();

    /**Construct the frame*/
    public TestFrame()
    {

```

```

        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    }
    try
    {
        jbInit();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}

/**Component initialization*/
private void jbInit() throws Exception
{
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("Test Windows Event");
}

/**Overridden so we can exit when window is closed*/
protected void processWindowEvent(WindowEvent e)
{
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING)
    {
        System.exit(0);
    }
}
}

```

The Java interpreter executes the project starting from the application class. The frame class creates the user interface and performs the actual operations.

NOTE: The validate method is in the Container class. It validates a container and all of its subcomponents. This method causes a container to layout its subcomponents again after the components it contains have been added to or modified. See Exercise 10.16.

The Application Class

The application class contains a constructor and a main method, as shown in Figure 10.36. The constructor creates an instance for the frame class and makes it visible. The main method is invoked by the Java interpreter to start the application class. Usually you do not need to change any code in the application class.

*****Insert Figure 10.36**

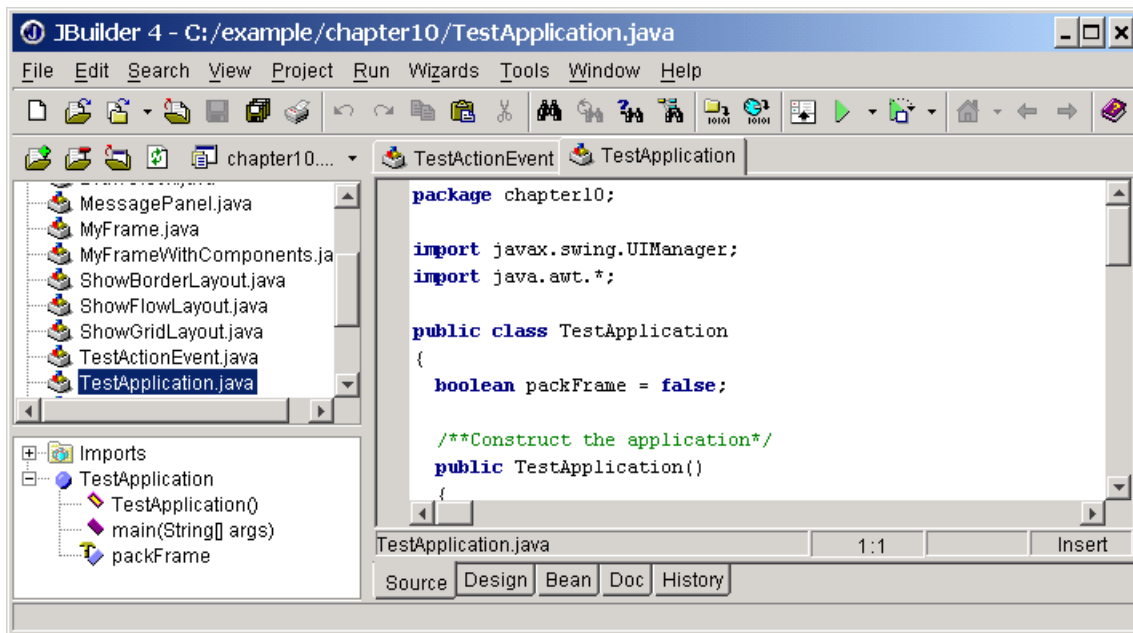


Figure 10.36

The application class contains a main method and a constructor.

NOTE: The `main` method also invokes the `setLookAndFeel` method in the `javax.swing.UIManager` class to set the GUI look and feel. Java supports three standard styles of look and feel: Windows, Motif, and Metal. The `getSystemLookAndFeelClassName()` method returns the name of the `LookAndFeel` class that implements the native system's look and feel if there is one; otherwise the name of the default cross platform `LookAndFeel` class is returned.

The Frame Class

You need to modify the frame class to write the appropriate code for the project. Before you modify the frame class, take a look at its contents. Highlighting `TestFrame.java` in the project pane displays the structure of the source code in the Structure pane and the source code itself in the Content pane (see Figure 10.37).

*****Insert Figure 10.37**

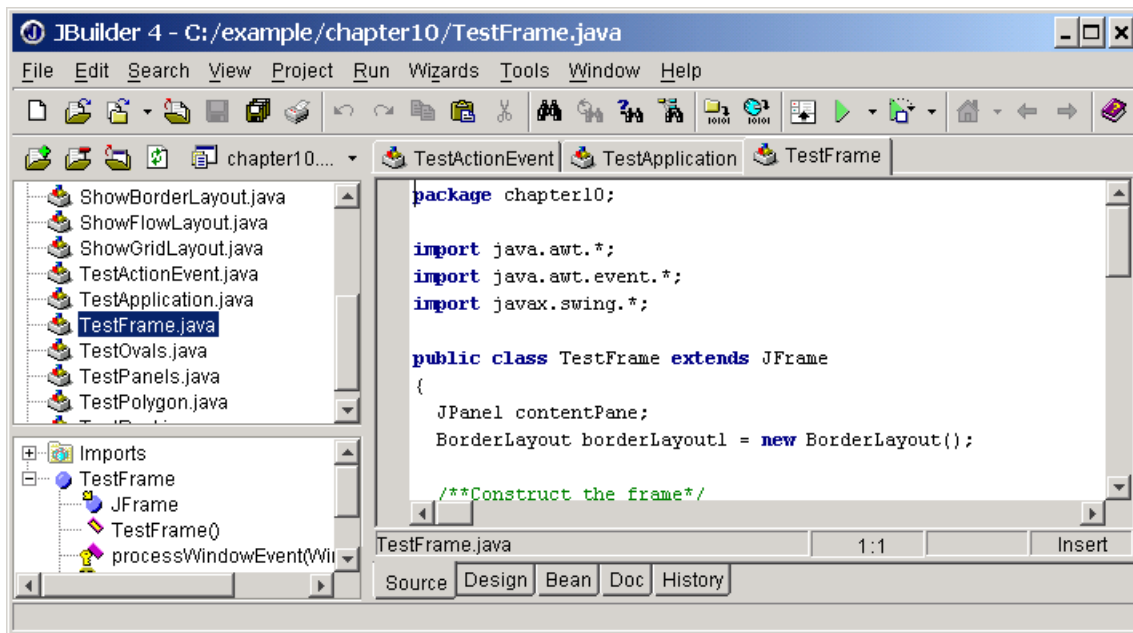


Figure 10.37

The frame class contains the code that creates user interface and carries out actual operations.

The `package` statement in the first line indicates that the program's bytecode file will be stored in the `c:\example\chapter10` directory. The `import` statements import standard Java packages. You can import additional packages if needed when you modify the code in the frame class. You can browse the imported classes by clicking the class in the Structure pane. For example, to browse `JFrame`, as shown in Figure 10.38, click `JFrame` in the Structure pane.

*****Insert Figure 10.38**

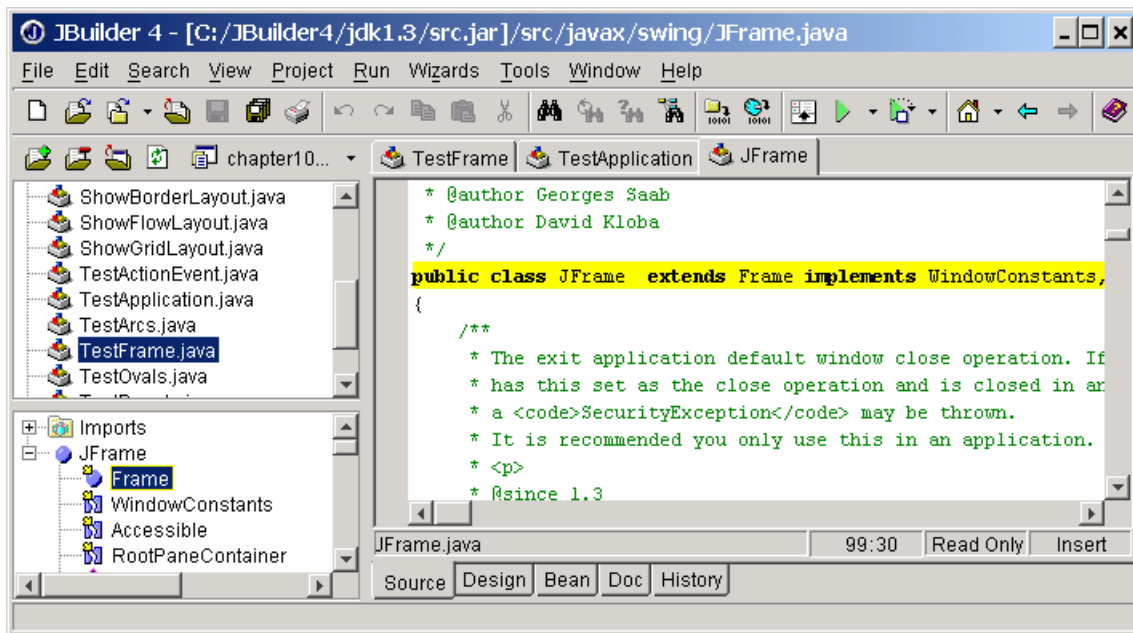


Figure 10.38

You can browse the Java files you imported to your class in the AppBrowser.

The `TestFrame` class extends `JFrame`. `TestFrame` has a method `jbInit()` and a constructor. The `jbInit()` method sets the frame's initial size, title, and the layout style. You also can modify or add new code in this method. The constructor simply invokes the `jbInit()` method.

The `processWindowEvent()` method is automatically generated by the Application wizard. This method is defined in `java.awt.Window`, which is a superclass of `JFrame`. It is invoked when a `WindowEvent` occurs. In `TestFrame`, the `processWindowEvent()` method invokes the same method defined in its superclass and exits the program when the window is closing. In JDK 1.3, you can simply use `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` to close the window.

If you ran the program now, you would see a blank frame.

Modifying the Code in the Frame Class

By now you know the files generated by the Application wizard as well as their contents. The Application wizard cannot generate everything you need in the project. To make the program work, you need to modify the `TestFrame` class as follows:

1. Implement the `ActionListener` interface, and the `actionPerformed` method as follows:

```

public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == jbtOk)
    {
        System.out.println("The OK button is clicked");
    }
    else if (e.getSource() == jbtCancel)
    {
        System.out.println("The Cancel button is clicked");
    }
}
}

```

2. Add the following declaration in the program:

```

// Create an object for "Close" button
private JButton jbtOk = new JButton("OK");
private JButton jbtCancel = new JButton("Cancel");

```

3. Add the following line in the `jbInit()` method to add buttons to the frame, and register the frame as a listener for buttons:

```

contentPane.setLayout(new FlowLayout());

// Add buttons to the frame
getContentPane().add(jbtOk);
getContentPane().add(jbtCancel);

// Register listeners
jbtOk.addActionListener(this);
jbtCancel.addActionListener(this);

```

The modified TestFrame is shown as follows:

```

// TestFrame.java: Modified TestFrame
package chapter10;

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

public class TestFrame extends JFrame implements ActionListener
{
    JPanel contentPane;
    BorderLayout borderLayout1 = new BorderLayout();

    // Create an object for "Close" button
    private JButton jbtOk = new JButton("OK");
    private JButton jbtCancel = new JButton("Cancel");

    /**Construct the frame*/
    public TestFrame()
    {

```

```

enableEvents(AWTEvent.WINDOW_EVENT_MASK);
try
{
    jbInit();
}
catch(Exception e)
{
    e.printStackTrace();
}
}

/**Component initialization*/
private void jbInit() throws Exception
{
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("Test Windows Event");

    contentPane.setLayout(new FlowLayout());

    // Add buttons to the frame
    getContentPane().add(jbtOk);
    getContentPane().add(jbtCancel);

    // Register listeners
    jbtOk.addActionListener(this);
    jbtCancel.addActionListener(this);
}

/**Overridden so we can exit when window is closed*/
protected void processWindowEvent(WindowEvent e)
{
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING)
    {
        System.exit(0);
    }
}

public void actionPerformed(ActionEvent e)
{
    /**@todo: Implement this java.awt.event.ActionListener method*/
    if (e.getSource() == jbtOk)
    {
        System.out.println("The OK button is clicked");
    }
    else if (e.getSource() == jbtCancel)
    {
        System.out.println("The Cancel button is clicked");
    }
}
}

```

Chapter Summary

In this chapter, you learned Java graphics programming using the container classes, UI component classes, and helper classes.

The container classes, such as JFrame, JPanel, and JApplet, are used to contain other components. The UI component classes, such as JButton, TextField, TextArea, JComboBox, JList, JRadioButton, and JMenu, are subclasses of JComponent. They are used to facilitate user interaction. These classes are referred to as Swing UI components and are grouped in the javax.swing package.

The helper classes, such as Graphics, Color, Font, FontMetrics, Dimension, and LayoutManager, are used by components and containers to draw and place objects. These classes are grouped in the java.awt package.

You learned to override the paintComponent method to display graphics and draw strings and simple shapes using the drawing methods in the Graphics class.

Java graphics programming is event-driven. The code is executed when events are activated. An event is generated by user actions, such as mouse movements, keystrokes, or clicking buttons. Java uses a delegation-based model to register listeners and handle events. External user actions on the source object generate events. The source object notifies listener objects of events by invoking the handlers implemented by the listener class.

Review Questions

- 10.1 Describe the Java graphics class hierarchy.
- 10.2 Describe the methods in Component, Frame, JFrame, JComponent, and JPanel.
- 10.3 Explain the difference between the AWT UI components, such as java.awt.Button, and the Swing components, such as javax.swing.JButton.
- 10.4 How do you create a frame? How do you set the size for the frame? How do you get the size of a frame? How do you add components to the frame?
- 10.5 Determine whether the following statements are true or false:
 - You can add a component to a button.
 - You can add a button to a frame.
 - You can add a frame to a panel.

- You can add a panel to a frame.
- You can add any number of components to a panel, a frame, or an applet.
You can derive a class from JPanel, JFrame, or JApplet.

10.6 Why do you need to use the layout managers? What is the default layout manager for the content pane of a frame? What is the default layout manager for a JPanel?

10.7 Can you use the setTitle method in a panel? What is the purpose of using a panel?

10.8 Describe FlowLayout. How do you create a FlowLayout manager? How do you add a component to a FlowLayout container? Is there a limit to the number of components that can be added to a FlowLayout container?

10.9 Describe GridLayout. How do you create a GridLayout manager? How do you add a component to a GridLayout container? Is there a limit to the number of components that can be added to a GridLayout container?

10.10 Describe BorderLayout. How do you create a BorderLayout manager? How do you add a component to a BorderLayout container? Can you add multiple components in the same section?

10.11 Suppose that you want to draw a new message below an existing message. Should the x, y coordinate increase or decrease?

10.12 How do you set colors and fonts in a graphics context? How do you find the current color and font style?

10.13 Describe the drawing methods for lines, rectangles, ovals, arcs, and polygons.

10.14 Write a statement to draw the following shapes:

- Draw a thick line from (10, 10) to (70, 30). You must draw several lines next to each other to create the effect of one thick line.
- Draw a rectangle of width 100 and height 50 with the upper-left corner at (10, 10).

- Draw a rounded rectangle with width 100, height 200, corner horizontal diameter 40, and corner vertical diameter 20.
- Draw a circle with radius 30.
- Draw an oval with width 50 and height 100.
- Draw the upper half of a circle with radius 50.
- Draw a polygon connecting the following points: (20, 40), (30, 50), (40, 90), (90, 10), (10, 30).
Draw a 3D cube like the one in Figure 10.39.

*****Insert Figure 10.39 (Same as Figure 8.31 in the 3rd Edition, p333)**

Figure 10.39

Use the drawLine method to draw a 3D cube.

- 10.15 Can a button generate WindowEvent? Can a button generate a MouseEvent? Can a button generate an ActionEvent?
- 10.16 Explain how to register a listener object and how to implement a listener interface.
- 10.17 What information is contained in an AWTEvent object and the objects of its subclasses? Find the variables, constants, and methods defined in these event classes.
- 10.18 How do you override a method defined in the listener interface? Do you need to override all the methods defined in the listener interface?
- 10.19 Describe the paintComponent method. Where is it defined? How is it invoked? Can you use the paintComponent method to draw things directly on a frame?

Programming Exercises

10.1 Write a program to meet the following requirements (see Figure 10.40):

- Create a frame and set its content pane's layout to FlowLayout.

- Create two panels and add the panels to the frame.
- Each panel contains three buttons. The panel uses FlowLayout.
- When a button is clicked, display a message on the console indicating that a button has been clicked. (Do this part after finishing the section "Event-driven Programming" in this chapter.)

*****Insert Figure 10.40**



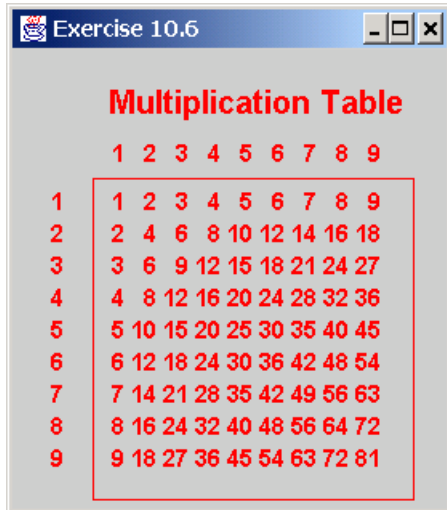
Figure 10.40

The first three buttons are placed in one panel, and the remaining three buttons are placed in another panel.

- 10.2 Rewrite the preceding program to create the same user interface. Instead of using FlowLayout for the frame's content pane, use BorderLayout. Place one panel in the south of the content pane and the other panel in the center of the content pane.
- 10.3 Rewrite the preceding program to create the same user interface. Instead of using FlowLayout for the panels, use GridLayout of two rows and three columns.
- 10.4 Rewrite the preceding program to create the same user interface. Instead of creating buttons and panels separately, define the panel class that extends the JPanel class. Place three buttons in your panel class, and create two panels from the user-defined panel class.
- 10.5 Write a program that uses the MessagePanel class in Example 10.5 to display a centered message in red, SansSerif, italic, and 20-point size.

10.6 Write a program that displays a multiplication table in a panel using the drawing methods, as shown in Figure 10.41.

***Insert Figure 10.41



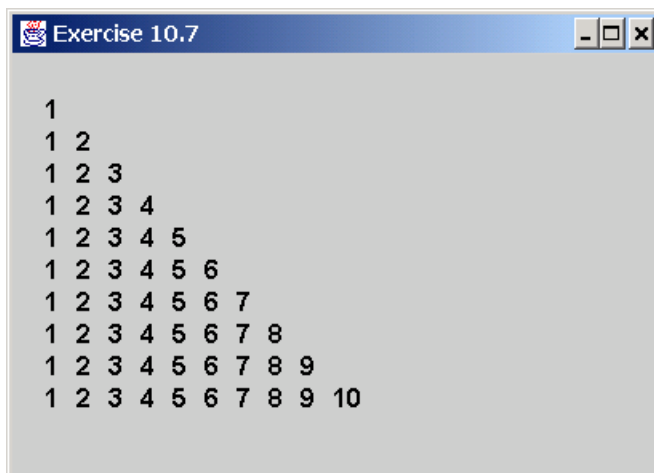
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Figure 10.41

The program displays a multiplication table.

10.7 Write a program that displays numbers, as shown in Figure 10.42. The number of lines in the display changes to fit the window as the window resizes.

***Insert Figure 10.42



```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Figure 10.42

The program displays a multiplication table.

10.8 Write a program that draws the diagram for the function $f(x) = x^2$. (see Figure 10.43).

Hint: Add points to a polygon `p` using the following loop:

```
double scaleFactor = 0.1;

for (int x=-100; x <= 100; x++)
{
    p.addPoint(x+200, 200-(int)(scaleFactor*x*x));
}
```

Use the `drawPolyline()` method in the `Graphics` class to connect the points.

***Insert Figure 10.43

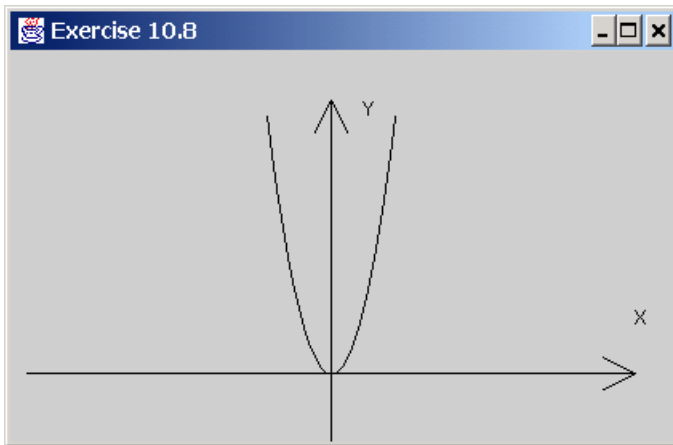


Figure 10.43

The program draws a diagram for function $f(x) = x^2$.

10.9 Write a program that draws the diagram for the sin function, as shown in Figure 10.44.

Hint: The Unicode for π is `\u03c0`. To display -2π , use `g.drawString("-2\u03c0", x, y)`. For a trigonometric function like $\sin(x)$, x is in radians. Use the following loop to add the points to a polygon `p`:

```
for (int x=-100; x <= 100; x++)
{
    p.addPoint(x+200,
        100-(int)(50*Math.sin((x/100.0)*2*Math.PI)));
}
```

-2π is at (100, 100), the center of axes is at (200, 100), and 2π is at (300, 100). Connect the points using `g.drawPolyline(p.xpoints, p.ypoints, p.npoints)` for a `Graphics` object `g`. `p.xpoints` returns an array of x coordinates, `p.ypoints` returns an array of y coordinates, and `p.npoints` returns the number of points in the `Polygon` object `p`.

***Insert Figure 10.44

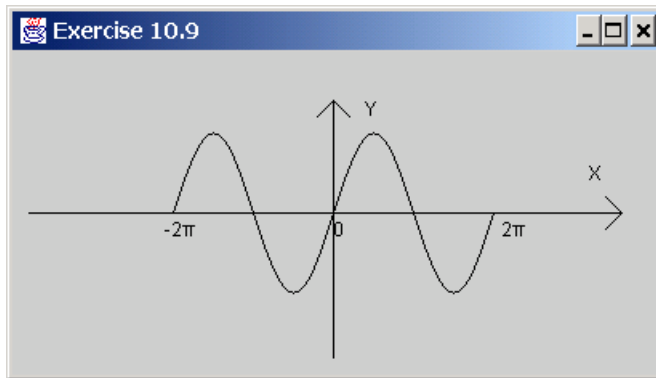


Figure 10.44

The program draws a diagram for function $f(x) = \sin(x)$.

10.10 Write a generic class that draws the diagram for a function. The class is defined as follows:

```
public abstract class AbstractDrawFunction extends JPanel
{
    /**Polygon to hold the points*/
    private Polygon p = new Polygon();

    /**Default constructor*/
    protected AbstractDrawFunction ()
    {
        drawFunction();
    }

    /**Return the y coordinate*/
    abstract double f(double x);

    /**Obtain points for x coordinates 100, 101, ..., 300*/
    public void drawFunction()
    {
        for (int x = -100; x <= 100; x++)
        {
            p.addPoint(x+200, 200-(int)f(x));
        }
    }

    /**Implement paintComponent to draw axes, labels, and
    *connecting points*/
    public void paintComponent(Graphics g)
    {
        // To be completed by you
    }
}
```

Test the class with the following functions:

```

_____  $f(x) = x^2;$ 
_____  $f(x) = \sin(x);$ 
_____  $f(x) = \cos(x);$ 
_____  $f(x) = \tan(x);$ 
_____  $f(x) = \cos(x) + 5\sin(x);$ 
_____  $f(x) = \cos(x) + 5\sin(x);$ 
_____  $f(x) = \log(x) + x^2;$ 

```

For each function, create a class that extends the AbstractDrawFunction class and implement the f method. Figure 10.45 displays the drawings for the sine function and the cosine function.

***Insert Figure 10.45

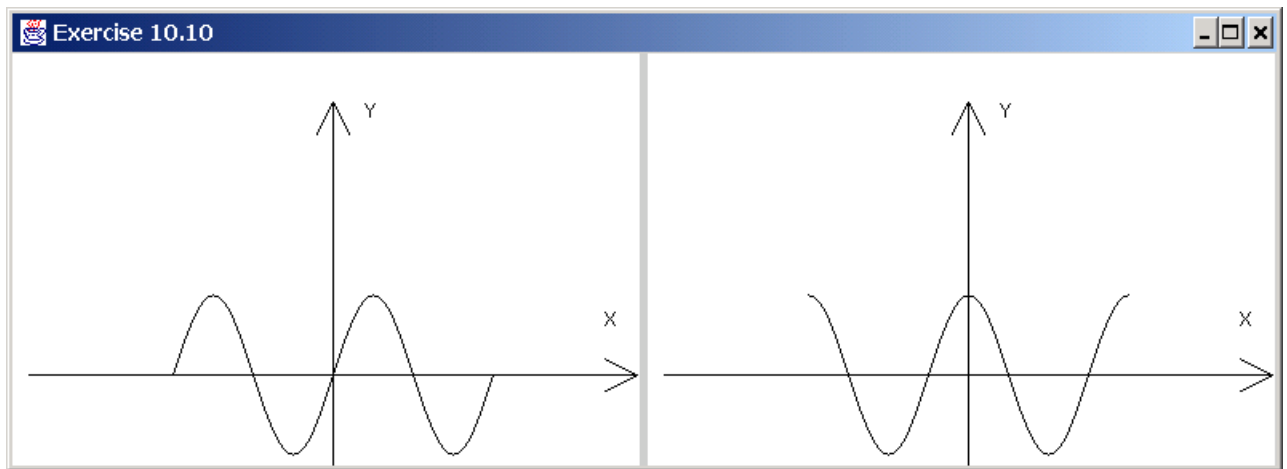


Figure 10.45

The drawings for the sine and cosine functions are displayed in a frame.

- 10.11 Write a program that draws a fan with four blades, as shown in Figure 10.46. Draw the circle in blue and the blades in red. To make the fan reusable, create a panel to display a fan, and place the panel in the frame. This panel can be reused elsewhere; for example, in Exercise 10.13. (Hint: Use the fillArc() method to draw the blades.)

***Insert Figure 10.46

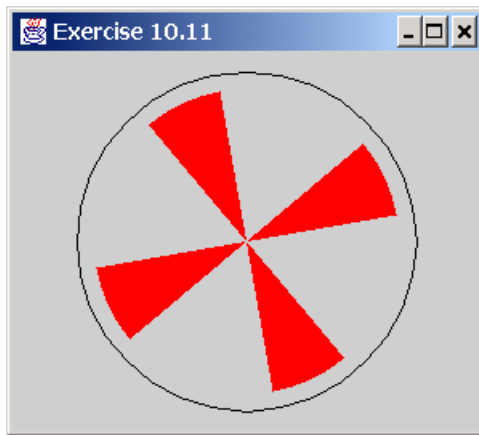


Figure 10.46

The drawing methods are used to draw a fan with four blades.

10.12 Modify Example 10.6 to draw the clock with more details on the hours and minutes, as shown in Figure 10.47.

*****Insert Figure 10.47**

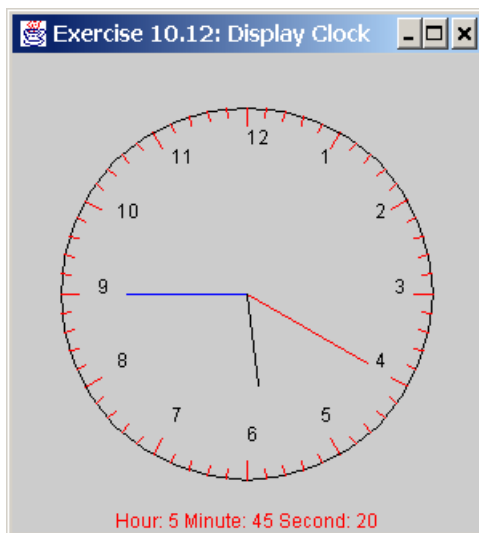


Figure 10.47

All the hours are displayed on the clock.

10.13 Write a program that places four fans in a frame of GridLayout with two rows and two columns, as shown in Figure 10.48.

*****Insert Figure 10.48**

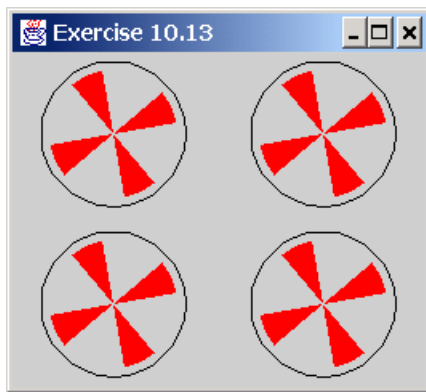


Figure 10.48

Four fans are placed in a frame.

- 10.14 Write a program that uses a pie chart to display the percentages of projects, quizzes, midterm exams, and final exam in the overall grade, as shown in Figure 10.49. Suppose that projects take 20% and displayed in red, quizzes take 10% and are displayed in green, midterm exams take 30% and are displayed in blue, and final exam takes 40% and is displayed in orange.

*****Insert Figure 10.49**

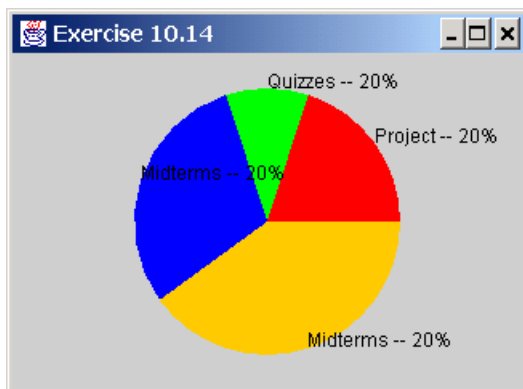


Figure 10.49

The pie chart displays the percentages of projects, quizzes, midterm exams, and final exam in the overall grade.

- 10.15 Write a program that creates four panels using the classes RectPanel, OvalsPanel, ArcsPanel, and PolygonsPanel presented in the section, "Drawing Geometric Figures," and places the panels in the content pane of the frame using a GridLayout, as shown in Figure 10.50.

***Insert Figure 10.50

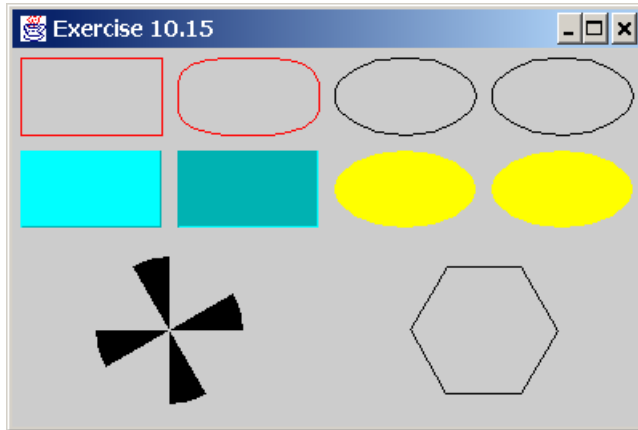


Figure 10.50

Four panels of geometric figures are displayed in a frame of the GridLayout.

10.16 Write a program with four buttons "Show Rectangle," "Show Oval," "Show Arc," and "Show Polygon." When you click a button, a corresponding panel (RectPanel, OvalsPanel, ArcsPanel, and PolygonsPanel) is added to the frame, as shown in Figure 10.51. By default, the RectPanel is displayed in the frame. (Hint: You may define a variable named figurePanel of the JPanel type to reference one of the four figure panels. Initially, figurePanel references an object of the RectPanel, and figurePanel is placed in the center of the frame's content pane. Whenever a new figure panel is selected, remove figurePanel from the content pane, assign the reference of the new figure panel to figurePanel, and add figurePanel to the content pane of the frame, repaint and validate the content pane as follows:

```
public void actionPerformed(ActionEvent e)
{
    getContentPane().remove(figurePanel);

    if (e.getSource() == jbtRect)
        figurePanel = rectPanel;
    else if (e.getSource() == jbtOval)
        figurePanel = ovalPanel;
    else if (e.getSource() == jbtArc)
        figurePanel = arcPanel;
    else if (e.getSource() == jbtPolygon)
        figurePanel = polygonPanel;

    getContentPane().add(figurePanel);
    getContentPane().repaint();
    getContentPane().validate();
}
```

***Insert Figure 10.51

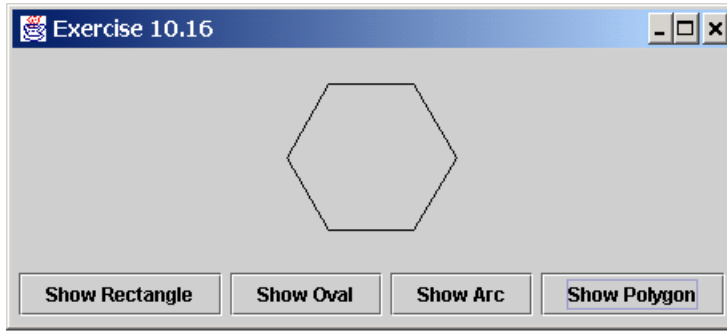


Figure 10.51

Clicking a button to add a corresponding panel to the frame.

10.17 Rewrite Example 10.5, "Using FontMetrics," to test MessagePanel from a main method inside MessagePanel as follows:

- Add a main method in the MessagePanel class. The main method creates a JFrame instance.
- Create and add a MessagePanel instance into the JFrame object. (Solution for this exercise can be found in **jb4EvenNumberedExercise\MessagePanel.java** in the CDROM.)

10.18 Rewrite Example 10.9, "Multiple Listeners for a Single Source," as follows:

- Create a method in TestMultipleListener as follows:

```
public void processButtons(ActionEvent e)
{
    if (e.getSource() == jbtOk)
    {
        System.out.println("The OK button is clicked");
    }
    else if (e.getSource() == jbtCancel)
    {
        System.out.println("The Cancel button is clicked");
    }
}
```

- Invoke processButtons(e) from the actionPerformed(e) method in TestMultipleListener.

- Modify SecondListener to invoke processButtons(e) defined in TestMultipleListener from the actionPerformed method in SecondListener. For the actionPerformed method in the SecondListener class to invoke the processButtons(e) method in the TestMultipleListener class, you may pass a reference of a TestMultipleListener object to SecondListener through the constructor of SecondListener.

10.19 In Exercise 10.14, you wrote a program that displays data in a pie chart. The program is difficult to reuse. In this exercise you will write a reusable component named PieChart to display a pie chart for any set of data. Suppose the data is stored in an array of double elements, named data, and the names for the data are stored in an array of strings, named dataName. For example, the enrollment data {200, 40, 50, 100, 40} stored in the array data are for {"CS", "Math", "Chem", "Biol", "Phys"} in the array dataName. The outline of this component is as follows:

```
public class BarChart extends JPanel
{
    /**Sample data, and data names*/
    private double[] data = {200, 140, 100, 60, 40};
    private String[] dataName = {"CS", "Math", "Chem", "Biol", "Phys"};

    /**Display the pie chart*/
    public void paintComponent(Graphics g)
    {
        // Write your code here
    }

    /**Set data*/
    public void setData(double[] newData)
    {
        data = newData;
        repaint();
    }

    /**Set data names*/
    public void setDataName(String[] newDataName)
    {
        dataName = newDataName;
        repaint();
    }
}
```

Hint: Each pie represents a percentage of the total data. Color the pie using the colors from an array

named `colors`, which is `{Color.red, Color.yellow, Color.green, Color.blue, Color.cyan, Color.magenta, Color.orange, Color.pink, Color.darkGray}`. Use `colors[i%colors.length]` for the *i*th pie. Use black color to display the data names. Figure 10.52 shows three pie charts created using this component.

***Insert Figure 10.52

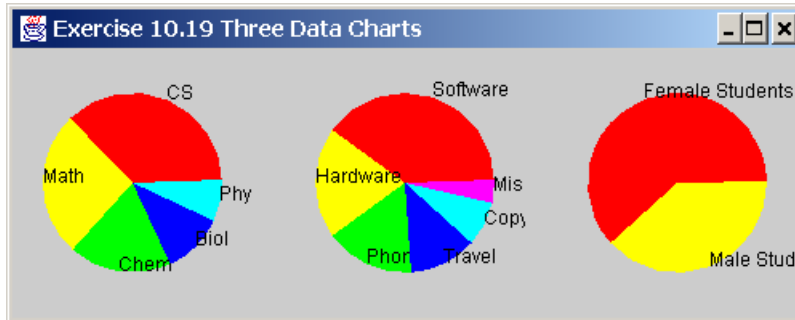


Figure 10.52

Three pie charts are placed in a frame of GirdLayout.

10.20 Similar to Exercise 10.19, create a new chart component named `BarChart` to display bar charts, as shown in Figure 10.53. Can you combine the `PieChart` and `BarChart` components into one component named `Chart`. Add a property named `chartType` to determine which type of chart is displayed.

***Insert Figure 10.53

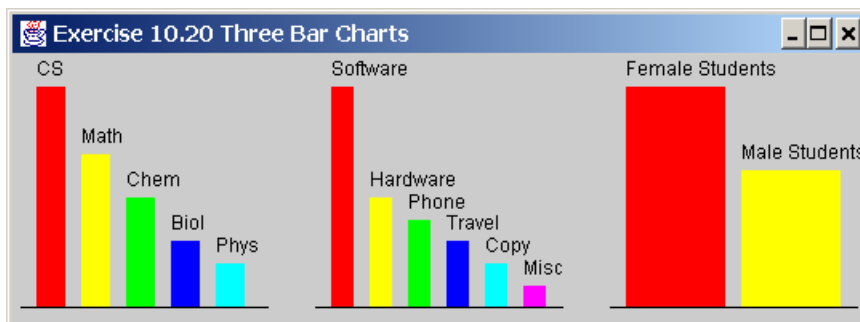


Figure 10.53

Three bar charts are placed in a frame of GirdLayout.