PEARSON NEW INTERNATIONAL EDITION

Java Foundations

John Lewis Peter DePasquale Joe Chase

Third Edition

ALWAYS LEARNING™

PEARSON®

Pearson New International Edition

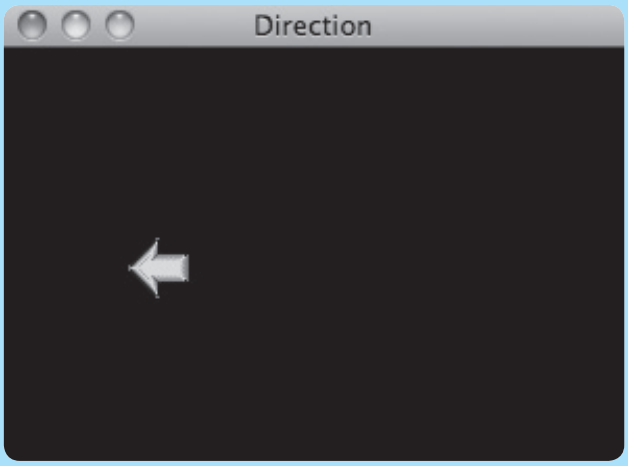
Java Foundations
John Lewis Peter DePasquale Joe Chase
Third Edition

PEARSON®

LISTING 27

```
//********************************************************************
//  Direction.java      Java Foundations
//
//  Demonstrates key events.
//********************************************************************

import javax.swing.JFrame;

public class Direction
{
   //-----------------------------------------------------------------
   //  Creates and displays the application frame.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Direction");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new DirectionPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

DISPLAY




Go to the end of the text for a full-color version of this figure.

LISTING 28

```
//********************************************************************
//  DirectionPanel.java       Java Foundations
//
//  Represents the primary display panel for the Direction program.
//********************************************************************

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

public class DirectionPanel extends JPanel
{
   private final int WIDTH = 300, HEIGHT = 200;
   private final int JUMP = 10; // increment for image movement

   private final int IMAGE_SIZE = 31;

   private ImageIcon up, down, right, left, currentImage;
   private int x, y;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this panel and loads the images.
   //-----------------------------------------------------------------
   public DirectionPanel()
   {
      addKeyListener (new DirectionListener());

      x = WIDTH / 2;
      y = HEIGHT / 2;

      up = new ImageIcon("arrowUp.gif");
      down = new ImageIcon("arrowDown.gif");
      left = new ImageIcon("arrowLeft.gif");
      right = new ImageIcon("arrowRight.gif");

      currentImage = right;

      setBackground(Color.black);
      setPreferredSize(new Dimension(WIDTH, HEIGHT));
      setFocusable(true);
   }
```

LISTING 28 *continued*

```
//-----------------------------------------------------------------
//  Draws the image in the current location.
//-----------------------------------------------------------------
public void paintComponent(Graphics page)
{
   super.paintComponent(page);
   currentImage.paintIcon(this, page, x, y);
}

//*****************************************************************
//  Represents the listener for keyboard activity.
//*****************************************************************
private class DirectionListener implements KeyListener
{
   //--------------------------------------------------------------
   //  Responds to the user pressing arrow keys by adjusting the
   //  image and image location accordingly.
   //--------------------------------------------------------------
   public void keyPressed(KeyEvent event)
   {
      switch (event.getKeyCode())
      {
         case KeyEvent.VK_UP:
            currentImage = up;
            y -= JUMP;
            break;
         case KeyEvent.VK_DOWN:
            currentImage = down;
            y += JUMP;
            break;
         case KeyEvent.VK_LEFT:
            currentImage = left;
            x -= JUMP;
            break;
         case KeyEvent.VK_RIGHT:
            currentImage = right;
            x += JUMP;
            break;
      }

      repaint();
   }

   //--------------------------------------------------------------
   // Provide empty definitions for unused event methods.
   //--------------------------------------------------------------
```

LISTING 28 *continued*

```
        public void keyTyped(KeyEvent event) {}
        public void keyReleased(KeyEvent event) {}
    }
}
```

image observer, the graphics context on which the image will be drawn, and the (x, y) coordinates where the image is drawn. An *image observer* is a component that serves to manage image loading; in this case we use the panel as the image observer.

The private inner class called `DirectionListener` is set up to respond to key events. It implements the `KeyListener` interface, which defines three methods that we can use to respond to keyboard activity. Figure 8 lists these methods.

Specifically, the `Direction` program responds to key pressed events. Because the listener class must implement all methods defined in the interface, we provide empty methods for the other events.

The `KeyEvent` object passed to the `keyPressed` method of the listener can be used to determine which key was pressed. In the example, we call the `getKeyCode` method of the `event` object to get a numeric code that represents the key that was pressed. We use a `switch` statement to determine which key was pressed and to respond accordingly. The `KeyEvent` class contains constants that correspond to the numeric code that is returned from the `getKeyCode` method. If any key other than an arrow key is pressed, it is ignored.

`void keyPressed (KeyEvent event)`
Called when a key is pressed.

`void keyReleased (KeyEvent event)`
Called when a key is released.

`void keyTyped (KeyEvent event)`
Called when a pressed key or key combination produces a key character.

FIGURE 8 The methods of the `KeyListener` interface

Key events fire whenever a key is pressed, but most systems enable the concept of *key repetition*. That is, when a key is pressed and held down, it's as if that key is being pressed repeatedly and quickly. Key events are generated in the same way. In the `Direction` program, the user can hold down an arrow key and watch the image move across the screen quickly.

The component that generates key events is the one that currently has the keyboard focus. Generally, the keyboard focus is held by the primary "active" component. A component usually gets the keyboard focus when the user clicks on it with the mouse. The call to the `setFocusable` method in the panel constructor sets the keyboard focus to the panel.

The `Direction` program sets no boundaries for the arrow image, so it can be moved out of the visible window and then moved back in if desired. You could add code to the listener to stop the image when it reaches one of the window boundaries. This modification is left as a programming project.

Extending Adapter Classes

In previous event-based examples, we've created the listener classes by implementing a particular listener interface. For instance, to create a class that listens for mouse events, we created a listener class that implemented the `MouseListener` interface. As we saw in the previous examples in this section, a listener interface often contains event methods that are not important to a particular program, in which case we provided empty definitions to satisfy the interface requirement.

An alternative technique for creating a listener class is to use inheritance and extend an *event adapter class*. Each listener interface that contains more than one method has a corresponding adapter class that already contains empty definitions for all of the methods in the interface. To create a listener, we can derive a new listener class from the appropriate adapter class and override any event methods in which we are interested. Using this technique, we no longer need to provide empty definitions for unused methods.

KEY CONCEPT

A listener class can be created by deriving it from an event adapter class.

The `MouseAdapter` class, for instance, implements the `MouseListener` interface and provides empty method definitions for the five mouse event methods (`mousePressed`, `mouseClicked`, and so on). Therefore, you can create a mouse listener class by extending the `MouseAdaptor` class instead of by implementing the `MouseListener` interface directly. The new listener class inherits the empty definitions and therefore doesn't need to define them.

Because of inheritance, we now have a choice when it comes to creating event listeners. We can implement an event listener interface, or we can extend an event adapter class. This is a design decision that should be considered carefully. Which technique is best depends on the situation.

5 Dialog Boxes

A component called a *dialog box* can be helpful in GUI processing. A dialog box is a graphical window that pops up on top of any currently active window so that the user can interact with it. A dialog box can serve a variety of purposes, such as conveying some information, confirming an action, or allowing the user to enter some information. Usually a dialog box has a solitary purpose, and the user's interaction with it is brief.

The Swing package of the Java class library contains a class called `JOptionPane` that simplifies the creation and use of basic dialog boxes. Figure 9 lists some of the methods of `JOptionPane`.

The basic formats for a `JOptionPane` dialog box fall into three categories. A *message dialog box* simply displays an output string. An *input dialog box* presents a prompt and a single input text field into which the user can enter one string of data. A *confirm dialog box* presents the user with a simple yes-or-no question.

Let's look at a program that uses each of these types of dialog boxes. Listing 29 shows a program that first presents the user with an input dialog box that requests the user to enter an integer. After the user presses the OK button on the input dialog box, a second dialog box (this time a message dialog box) appears, informing the user whether the number entered was even or odd. After the user dismisses that box, a third dialog box appears, to determine whether the user would like to test another number. If the user presses the button labeled Yes, the series of dialog boxes repeats. Otherwise, the program terminates.

The first parameter to the `showMessageDialog` and the `showConfirmDialog` methods specifies the governing parent component for the dialog box. Using a null reference as this parameter causes the dialog box to appear centered on the screen.

```
static String showInputDialog (Object msg)
    Displays a dialog box containing the specified message and an input text field.
    The contents of the text field are returned.

static int showConfirmDialog (Component parent, Object msg)
    Displays a dialog box containing the specified message and Yes/No button
    options. If the parent component is null, the box is centered on the screen.

static void showMessageDialog (Component parent, Object msg)
    Displays a dialog box containing the specified message. If the parent
    component is null, the box is centered on the screen.
```

FIGURE 9 Some methods of the `JOptionPane` class